Universitat politècnica de catalunya (UPC) - BarcelonaTech

Facultat d'informàtica de Barcelona (FIB)

# Checkpoint and restore of Singularity containers

Grado en ingeniería informática

Tecnologías de la Información

Memoria

25/04/2019

*Director:*
Jordi Guitart Fernandez

*Autor:*
Enrique Serrano Gómez

*Departament:*
Arquitectura de Computadors

**Abstract**

Singularity es una tecnología de contenedores software creada según las necesidades de científicos para ser utilizada en entornos de computación de altas prestaciones.

Hace ya 2 años desde que los usuarios empezaron a pedir una integración de la funcionalidad de *Checkpoint/Restore*, con CRIU, en contenedores Singularity. Esta integración ayudaría en gran medida a mejorar la gestión de los recursos computacionales de las máquinas. Permite a los usuarios guardar el estado de una aplicación (ejecutándose en un contenedor Singularity) para poder restaurarla en cualquier momento, sin perder el trabajo realizado anteriormente. Por lo que la posible interrupción de una aplicación, debido a un fallo o voluntariamente, no es una pérdida de tiempo de computación.

Este proyecto muestra como es posible realizar esa integración.

Singularity és una tecnologia de contenidors software creada segons les necessitats de científics, per ser utilitzada a entorns de computació d'altes prestacions.

Fa 2 anys desde que els usuaris van començar a demanar una integració de la funcionalitat de *Checkpoint/Restore*, amb CRIU, a contenidors Singularity. Aquesta integració ajudaria molt a millorar la gestió dels recursos computacionals de les màquines. Permet als usuaris guardar l'estat d'una aplicació (executant-se a un contenidor Singularity) per poder restaurar-la en qualsevol moment, sense perdre el treball realitzat anteriorment. Per la qual la possible interrupció d'una aplicació, a causa de una fallada o voluntàriament, no és una pèrdua de temps de computació.

Aquest projecte mostra com és possible realitzar la integració.

Singularity is a software container technology created thinking in the needs of the scientists, so it can be used on High Performance Computing environments.

It has been 2 years since the users started demanding an integration of the Checkpoint/Restore functionality, using CRIU on Singularity containers. This integration would help to a large degree to improve the management of the machine's computational resources. It allows the users to save the state of an application (being executed on a Singularity container) to be able of restoring it anytime, without losing the work already done. So that the prospect of interrupting an application, due to an error or voluntarily, is not a computation time lost.

This project shows how the integration can be done.

# Índice de contenidos

# Índice de tablas

# Índice de figuras

# 1 Contexto

El objeto de estudio de este proyecto es la tecnología de Contenedores Software, en concreto Singularity y la tecnología *checkpoint/restore*.

Las aplicaciones dependen de la configuración del entorno y de librerías específicas. Esto provoca que la portabilidad de las aplicaciones sea complicada, debido a que cada máquina tiene que tener la configuración apropiada del entorno y librerías pertinentes instaladas. Aún teniendo la misma configuración y librerías en dos maquinas diferentes, es difícil conseguir que sean entornos idénticos.

Un contenedor en el que está la aplicación instalada, contiene la configuración del entorno, las librerías, dependencias y archivos necesarios para ejecutar la aplicación. Todo esto de manera independiente a la configuración y entorno del sistema anfitrión. De hecho, se puede pensar en el contenido de la imagen de un contenedor como una instalación de una distribución Linux.[1][2][3]

Por lo tanto, solo se necesita el contenedor para ejecutar la aplicación deseada y puedes estar seguro de que siempre que ejecutes esa aplicación contenida, aunque sea en diferentes máquinas, estás ejecutando la aplicación en un entorno idéntico.

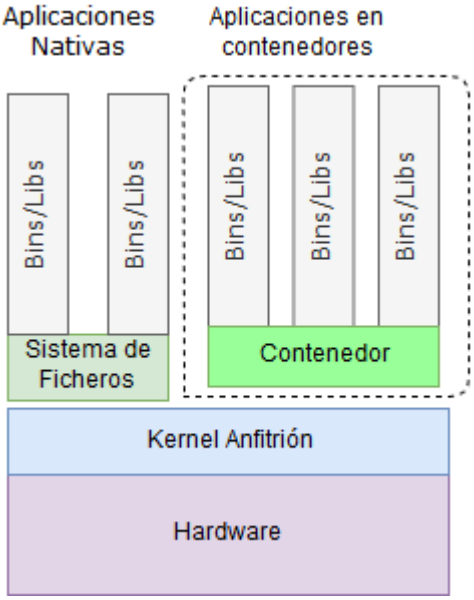


Figura 1: *Arquitectura Contenedor Software*

Los contenedores comparten el mismo núcleo del sistema operativo anfitrión y aíslan los procesos de la aplicación del resto del sistema. No virtualiza hardware ni carga otro sistema operativo como haría una máquina virtual por lo que es una tecnología mucho más ligera que una máquina virtual (MV).[4]

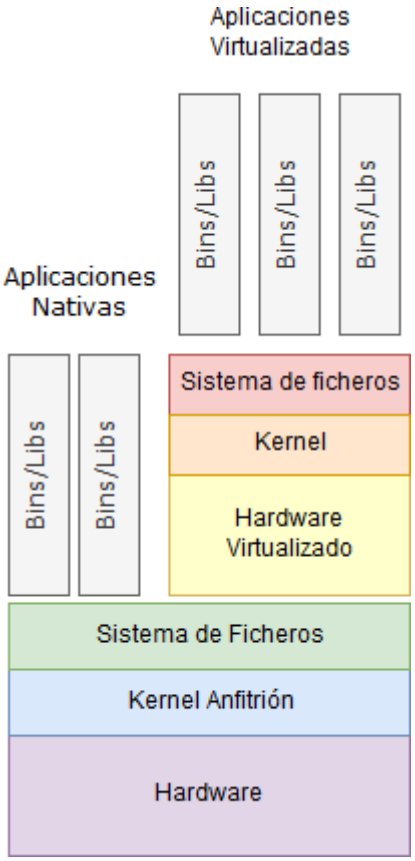


Figura 2: *Arquitectura Máquina virtual*

Esta tecnología se puede aplicar a problemas donde características como la portabilidad, configurabilidad y el aislamiento son necesarias.

## 1.1 Uso de Contenedores

Como las MVs, los contenedores pueden ser usados de multitud de maneras. Podemos diferenciar las siguientes categorías:

- Contenedores de aplicaciones, incluyen los componentes críticos de la aplicación junto con micro servicios de una aplicación modular, multinivel.
- Contenedores de sistema, esencialmente MVs en contenedores que combinan la versatilidad de entornos de sistemas personalizados con la eficiencia de recursos de los contenedores.
- Contenedores 'Pet', aplicaciones monolíticas completamente empaquetadas o entornos de múltiples aplicaciones.

- Contenedores *Super privileged*(SPC), estos tienen privilegios de sistema elevados que son requeridos para el núcleo de infraestructuras de servicios como *backup*, monitorización y seguridad.

Las aplicaciones en contenedores más comunes incluyen:

- *Web proxies* y Application Delivery Controllers, ADCs (NGINX y HAProxy).
- *Logging Software* (fluentd).
- Motores de búsqueda (Elastic).
- Bases de datos y cachés (MongoDB, PostgresSQL,etc).
- Plataformas de aplicaciones, servicios (JVM, formularios PHP, Consul).

[5]

## 1.2 Singularity

Singularity es una solución de contenedores creada debido a la necesidad de los científicos de tener una tecnología de contenedores con un conjunto de funcionalidades específico y apropiado para ellos. Esta desarrollada teniendo en mente los entornos HPC.

Singularity ofrece soporte a los existentes y tradicionales recursos HPC de una manera fácil, simplemente instalando un solo paquete en el sistema operativo anfitrión. Se pueden ejecutar imágenes en sistemas tan antiguos como Linux 2.2.

Además, da soporte de forma nativa a Infiniband, Lustre, y trabaja sin problemas con todos los gestores de recursos. También tiene incorporado soporte para MPI y para contenedores que necesitan balancear recursos de GPU.[6]

A diferencia de otras tecnologías de contenedores, Singularity no tiene la funcionalidad para hacer *checkpoint/restore* de sus contenedores y esta función es ampliamente deseada dentro de la comunidad, ya que es una mejora muy grande en el campo de *Mobility of Compute*.[7]

## 1.3 Actores implicados

Para el desarrollo de este proyecto tenemos como actores principales, al director del proyecto, el desarrollador y en menor medida, los desarrolladores de Singularity, que mostraron su interés en el proyecto y ofrecieron su ayuda en la medida de lo posible.

Los beneficiados de la realización del proyecto serán todos los usuarios/desarrolladores de aplicaciones las cuales se beneficien del uso de contenedores Sigularity, aplicaciones de HPC, etc. Todo debido a que el añadido de esta funcionalidad puede hacer que gente que no utilizaba contenedores Singularity tenga una motivación extra para empezar a usarlos y los ya usuarios de Singularity van a poder trabajar en sus proyectos con esta nueva funcionalidad.

# 2 Estado del arte

Este proyecto se centra en la integración de la funcionalidad de *checkpoint/restore*. A medida que la popularidad de la tecnología de contenedores ha ido incrementando, la necesidad de esta funcionalidad lo ha hecho también.

## 2.1 *Checkpoint*

Se refiere a guardar, en el medio de almacenamiento, el estado de una aplicación en ejecución de manera que se pueda cargar ese estado y la aplicación se ejecute como si nada hubiera pasado. Cuando una aplicación falla y deja de funcionar, se puede leer el último estado guardado de la aplicación y restaurarse perdiendo el mínimo tiempo posible y como si nada hubiese pasado.

El factor que afecta más al rendimiento de un sistema de *checkpoint* es el tiempo añadido por el *checkpoint* y reducir el tiempo necesario es un tema muy investigado.[8]

En un escenario típico de uso, se realizan *checkpoint* de la aplicación frecuentemente, para que la tolerancia a errores sea mayor y en caso de fallo poder restaurar la aplicación desde un momento más cercano en el tiempo en lugar de empezar desde el principio.

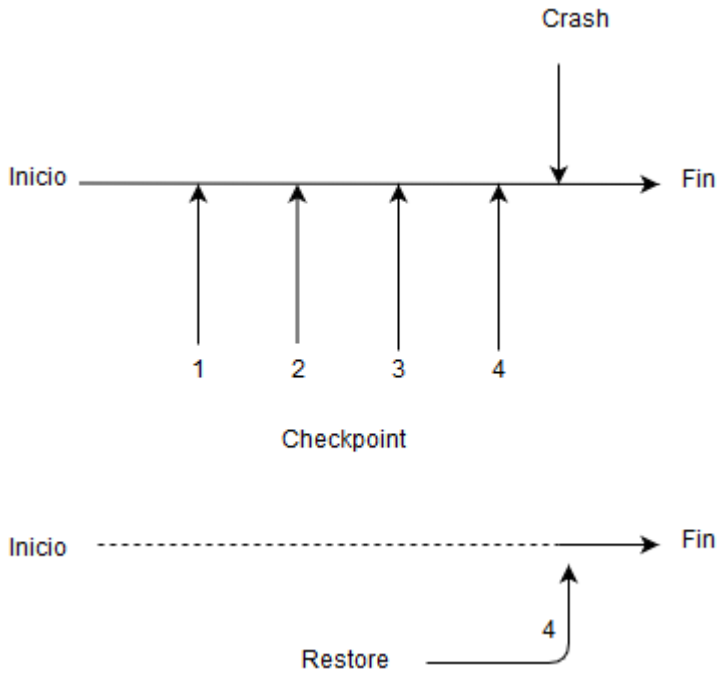


Figura 3: *Escenario típico de uso de Checkpoint/Restore*

## 2.2 Sistemas de *Checkpoint*

Los sistemas de *checkpoint* se pueden clasificar en *checkpoint* a nivel de usuario y *checkpoint* a nivel de sistema dependiendo el mecanismo de implementación.

- *Checkpoint* a nivel de usuario. Es la implementación *Checkpoint and Rollback Recovery (CRR)* del programa objetivo en espacio de usuario, el cual no necesita modificar el núcleo del sistema operativo. Este tipo de *checkpoint* se puede clasificar en 2 tipos según el mecanismo de implementación:
  - *Application-assisted checkpoint*, necesita modificar el código fuente de la aplicación, definiendo los contenidos que se tienen que guardar. Al realizar el *checkpoint*, solo los contenidos predefinidos se guardarán.
  - *Checkpoint* basado en el entorno de librerías, realizan el CRR del programa a mediante el enlace de librerías. Al compilar la aplicación, el entorno de librerías ofrecido por el sistema de *checkpoint* se enlaza dinámicamente y no necesita modificar el código fuente de la aplicación.
- *Checkpoint* a nivel de sistema, el cual realiza la función CRR del programa mediante modificación del sistema operativo. Puede ser clasificado en dos tipos, de acuerdo con el mecanismo de implementación:
  - Realiza la función CRR mediante el añadido de llamadas a sistema.
  - Realiza la función CRR aprovechando el mecanismo de *loadable kernel mechanism (LKM)* del sistema operativo.

[8]

## 2.3 *Container Checkpoint*

Se hace *Checkpoint* de un grupo de procesos que están en un entorno aislado del sistema anfitrión. El tipo de *Checkpoint* requerido es a nivel de sistema. Recursos de los procesos y las relaciones entre estos recursos tienen que ser guardados.

Los recursos del contenedor son gestionados y aislados por componentes, por ejemplo, recursos de Linux Containers(LXC) son gestionados por *Control Groups* (Cgroups) y aislados por el espacio de nombres. La estructura del árbol de procesos, los recursos del proceso y los recursos compartidos han de ser guardados de manera que puedan ser restaurados en cualquier momento.[8]

## 2.4 *Checkpoint/Restore in Userspace, CRIU*

Es una herramienta de software para el sistema operativo Linux. Actualmente está integrado en contenedores tipo LXC, OpenVZ, Docker. Permite realizar el *checkpoint/restore* de una manera totalmente transparente al sistema operativo. La ventaja es que no tiene pre-requisitos para poder usarse. No requiere de librerías especiales enlazadas a la aplicación o entornos especialmente preparados para interceptar llamadas a sistema.

Para alcanzar el nivel de transparencia de CRIU, se necesitaba hacer algunos cambios/añadidos en el núcleo de Linux. Estas modificaciones fueron aceptadas por la comunidad del núcleo de Linux ya que también podian utilizarse en otros casos en los que una mayor información sobre los procesos en ejecución sería útil.

Por eso, aunque se llama *Checkpoint/Restore in Userspace*, realmente es en espacio de usuario y núcleo, y pertenecería al *checkpoint* a nivel de sistema.[9]

Con esta herramienta, podemos congelar una aplicación en ejecución, o parte de ella, y hacer un *checkpoint* en un conjunto de archivos guardados en disco. Estos archivos se utilizan para restaurar la aplicación y ejecutarla exactamente en el estado que se encontraba.[10]

Aún hay varias cosas de las que no se puede hacer *checkpoint* y hay ciertos campos que pueden verse alterados después de un ciclo de *checkpoint/restore*. Es una herramienta que tiene limitaciones y la integración con las tecnologías de contenedores no es perfecta, pero aun así, es una herramienta muy útil para desarrollar esta funcionalidad y por la que se han decantado otras tecnologías de contenedores para hacerlo.[11][12][13]

## 2.5 Conclusiones

La funcionalidad de *checkpoint/restore* tiene un comportamiento y mecanismos de actuación claros. Además, el auge de los contenedores software y su uso en sistemas distribuidos hace que sea necesaria la investigación y desarrollo de esta tecnología.

La limitación más clara, además del tiempo añadido por *checkpoints* recurrentes, es la portabilidad de ciertas aplicaciones entre diferentes máquinas. Esto último es necesario para poder mover el mismo contenedor entre máquinas con libertad, sin perder su estado y tiempo invertido en la computación de sus tareas.

La funcionalidad de *checkpoint/restore* ha sido implementada en contenedores LXC, Docker. Estos últimos, teniendo la funcionalidad desarrollada utilizando CRIU y otra implementación nativa que está en fase *beta*, y desarrollando el siguiente paso, *live migration*.

Por lo que con nuestro proyecto, también buscamos ampliar el conocimiento sobre *checkpoint/restore* de contenedores software.

# 3 Formulación del problema

Tener la capacidad de hacer *checkpoint/restore* es una mejora sustancial a la capacidad computación general. A las ventajas de los contenedores se le añade la flexibilidad que otorga el *Checkpoint/Restore*.

Uno de los problemas actuales en HPC es que puede ser difícil de predecir con antelación que cantidad de recursos que requerirá una aplicación especifica, ya que el tiempo de computación y los recursos disponibles son limitados.

Permitiría que tareas con una necesidad de computación menor, o menor prioridad, puedan ser reemplazadas por otras tareas que necesiten más recursos. Una vez estas tareas finalicen o dejen de necesitar tantos recursos, las tareas a las que se les hizo *checkpoint* pueden ser restauradas y que continúen desde el punto en que lo dejaron. De esta manera no se pierde tiempo de computación y se pueden utilizar los recursos de las máquinas bajo demanda.

Por ejemplo, una tarea A está utilizando un 20% de potencia de cálculo durante 3 días y necesitamos ejecutar una tarea B de alta prioridad que necesita el 100%. Haríamos *checkpoint* de la tarea A, ejecutaríamos la tarea B y una vez termine, restauramos la anterior tarea A sin perder los 3 días de ejecución.

Este es un paso importante para lograr migrar aplicaciones contenidas entre máquinas de la nube. De esta manera, tienes a tu disposición no solo los recursos de la máquina anfitrión que ha puesto en marcha el contenedor, sino los recursos de toda la nube. Un paso más en el camino para poder conseguir una *Mobility of Compute* total en la que las tareas pueden cambiar de máquina bajo demanda gracias a la combinación de contenedores y la tecnología de *Checkpoint/Restore*.

Singularity no dispone de la funcionalidad para realizar *checkpoint/restore* de sus contenedores. Esta funcionalidad fue pedida a principios de enero de 2017 a los desarrolladores de Singularity y apareció en una presentación que hizo Singularity a finales de 2017 en su hoja de ruta.[7][3]

Como hemos comentado, otras tecnologías de contenedores como *Docker* y *Linux Containers (LXC)*, han desarrollado esta funcionalidad. Singularity tendrá dificultades técnicas añadidas debido a las diferencias con estas tecnologías de contenedores.[14]

# 4 Planteamiento inicial

En esta sección vamos a mostrar el planteamiento inicial que tuvo el proyecto para poder compararlo con el planteamiento final que veremos en la siguiente sección.

## 4.1 Alcance

El alcance del proyecto será realizar *checkpoint/restore* de un contenedor Singularity. Las funciones que podrá realizar el contenedor para poder ser objeto de *checkpoint/restore* serán limitadas debido a la complejidad de la funcionalidad y el tiempo límite del proyecto.

Una vez desarrollada la funcionalidad, una posible demostración de su uso sería, por ejemplo, una aplicación que escriba por pantalla una serie de números. Realizaríamos *checkpoint/restore* y veríamos como la aplicación sigue escribiendo la secuencia de números desde el último número que vimos antes de hacer *checkpoint*.

### 4.1.1 Posibles obstáculos

El desarrollo inicialmente quiere integrar la tecnología de *Checkpoint/Restore In Userspace* (CRIU) en Singularity, como hicieron otras tecnologías de contenedores como Docker. El caso es que Singularity tiene algunas características que podrían dificultarlo, como las siguientes:

- El usuario dentro del contenedor es el mismo usuario que fuera de él. Esto podría ser un problema debido a que es una base del funcionamiento de Singularity y la ejecución de *checkpoint/restore* necesita privilegios de root, debido a que necesita acceso a estructuras de sistema.

- No dispone de un *daemon* con privilegios de root. Docker utiliza este *daemon* como punto de partida para realizar el *checkpoint*, habrá que buscar una alternativa para Singularity

Esto podría significar que la integración fuera más costosa o incluso que no resultara provechoso integrar CRIU debido a las limitaciones que pudiera imponer a los contenedores. En este caso desarrollar la funcionalidad de manera nativa en Singularity pasaría a ser la opción a seguir.

El límite de tiempo es el mayor obstáculo que nos encontramos debido al esfuerzo necesario para realizar el proyecto y los obstáculos descritos. Contamos con una metodología y planificación específica para que el proyecto se lleve a cabo en el tiempo establecido.

## 4.2 Metodología y rigor

Para la realización de este proyecto seguiremos un método de trabajo iterativo. Al ser un único desarrollador y no saber todos los requisitos de antemano es la opción más sencilla y funcional. El desarrollo se conformara de las siguientes fases de manera iterativa hasta que se hayan finalizado todas las funcionalidades que requiere el objetivo.

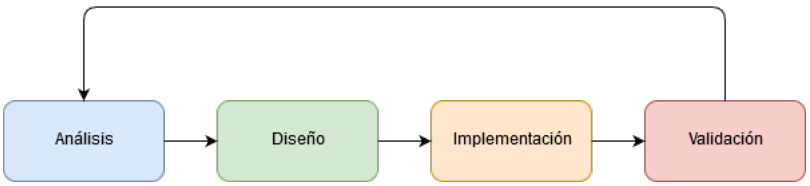


Figura 4: *Método iterativo.*

Además, se mantendrán reuniones periódicas con el tutor a medida que vaya avanzando el proyecto.

### 4.2.1 Herramientas

Para la gestión del código utilizaremos la tecnología de control de versiones *Git*. Dado que el código fuente sobre el que trabajaremos esta alojado en *GitHub*, usaremos este mismo servicio para alojar nuestro repositorio.

El trabajo se realizara siguiendo el *Forking Workflow* y *Feature Branch Workflow*. Trabajaremos sobre una copia del repositorio original de Singularity y crearemos ramas de desarrollo para cada funcionalidad. No se sigue un modelo centralizado porque, un método de trabajo basado en diferentes ramas para cada funcionalidad es una manera más organizada de llevar a cabo el trabajo. De esta manera tenemos la ventaja de poder tener una rama principal con un código correcto y funcional, sobre la que poder crear nuevas ramas de desarrollo sin depender de la finalización de otras funcionalidades en proceso.

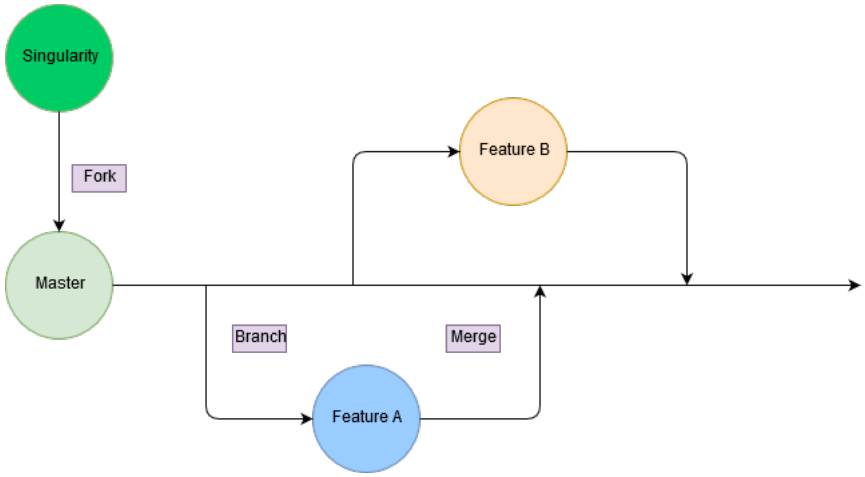


Figura 5: Diagrama de flujo de trabajo.

Para la edición de la documentación se utilizará *Overleaf*, editor LaTeX online. Para la gestión de documentación extra y redundancia con los documentos de *Overleaf*, utilizaremos *Dropbox*. Para la realización de diagramas utilizaremos la herramienta *Draw.io*.

### 4.2.2 Método de validación

La validación será realizada mediante pruebas locales para cada funcionalidad. Habrá comunicación con los desarrolladores de Singularity para comprobar que el desarrollo y las pruebas que se llevan a cabo son correctas.

## 4.3 Planificación

El proyecto tiene una duración estimada de 4 meses, desde el 15 de septiembre hasta el 15 de enero y desde este día hasta el día de presentación. Con una dedicación de 22 horas a la semana, estimamos una duración aproximada de 400 horas.

### 4.3.1 Descripción de las fases y tareas

Como describimos en el apartado de metodología y recursos, seguiremos una metodología iterativa que se verá reflejada en la elección y disposición en el tiempo de las tareas. Cada tarea seguirá el mismo ciclo de proceso: análisis, diseño, implementación y validación. Aun así se realizara una fase de análisis global antes de empezar con la dinámica de trabajo descrita.

El proyecto esta dividido en 5 fases: Planificación inicial, Análisis del proyecto, Estructuración Singularity, Integración CRIU-Singularity, Finalización del proyecto.

En paralelo a estas fases se realizará un borrador de la documentación del proyecto, además de reuniones de seguimiento con el tutor.

#### 4.3.1.1 Planificación inicial

En esta primera fase se realiza la planificación inicial del proyecto, que incluye la documentación y el análisis del proyecto. Esto nos permite definir los objetivos y la metodología que, junto con la planificación, nos permitan llevarlos a cabo.

#### 4.3.1.2 Análisis Proyecto

Se realiza un estudio de la implementación existente de Singularity y CRIU. La finalidad es la de tener una base sólida de conocimientos sobre el funcionamiento y estructura de estos, lo que nos permitirá encarar las fases de diseño e implementación con garantías.

En esta fase se analizaran también los obstáculos y soluciones que encontraron los desarrolladores de Docker al realizar la integración de CRIU, ya que, aun habiendo diferencias entre Singularity y Docker, es bastante probable que los problemas que encontraron ellos nos los encontremos también.

#### 4.3.1.3 Estructuración Singularity

Hemos de preparar las estructuras de Singularity para que la información necesaria para realizar un *checkpoint* sea accesible y además prepara la localización desde la cual se utilizará CRIU y podrá acceder a estos datos. Es el paso previo necesario a la integración de CRIU en Singularity.

#### 4.3.1.4 Integración CRIU-Singularity

Esta tarea es la más importante del proyecto. Tendremos que modificar, adaptar Singularity y, quizás, CRIU para que sea posible hacer *checkpoint* y *restore* de los contenedores Singularity. Singularity tendrá CRIU integrado de manera que realizar el *checkpoint* sea posible internamente.

#### 4.3.1.5 Finalización proyecto

Esta fase se llevará a cabo una vez la parte técnica del proyecto este terminada. Se realizará una evaluación del proyecto, en concreto evaluación de nuestra implementación de *Checkpoint/Restore* para saber que tipo de aplicaciones son compatibles con ella.

Además, se escribirá la memoria a partir del borrador que hemos realizado a lo largo del desarrollo y prepararemos la presentación del proyecto ante el tribunal.

| Tareas | Horas | Dependencia |
|---|---|---|
| **1. Planificación inicial** | **70** | **-** |
| 1.1. Alcance y contexto | 25 | - |
| 1.2. Planificación temporal | 10 | 1.1 |
| 1.3. Presupuesto y sostenibilidad | 15 | 1.2 |
| 1.4. Documentación final hito inicial | 20 | 1.3 |
| **2. Análisis del proyecto** | **70** | **-** |
| 2.1. Estudio implementación Singularity | 25 | - |
| 2.2. Estudio implementación CRIU | 15 | - |
| 2.3. Estudio implementación CRIU-Docker | 10 | - |
| 2.4. Estudio implementación Checkpoint | 20 | - |
| **3. Estructuración Singularity** | **85** | **2.4** |
| 3.1. Análisis | 15 | 2.4 |
| 3.2. Diseño | 20 | 3.1 |
| 3.3. Implementación | 40 | 3.2 |
| 3.4. Validación | 10 | 3.3 |
| **4. Integración CRIU-Singularity** | **120** | **3.4** |
| 4.1. Análisis | 20 | 3.4 |
| 4.2. Diseño | 20 | 4.1 |
| 4.3. Implementación | 65 | 4.2 |
| 4.4. Validación | 15 | 4.3 |
| **5. Finalización del proyecto** | **50** | **4.4** |
| 5.1. Evaluación | 10 | 4.4 |
| 5.2. Memoria | 20 | 5.1 |
| 5.3. Presentación | 20 | 5.2 |
| **Total** | **400** | **-** |

Tabla 1: Horas por tarea y sus dependencias.

## 4.4 Recursos

Para llevar a cabo el proyecto nos valdremos de diferentes recursos. Los recursos humanos consisten en la persona que realizará el proyecto, la cual se encargará de llevar a cabo todas las tareas descritas.

Los recursos materiales, en nuestro caso *hardware* y *software* son los siguientes:

- Ordenador sobremesa y sus periféricos, con sistema operativo Ubuntu 18.04 LTS.
- Ordenador portátil con Ubuntu 18.04 LTS, de manera que podamos trabajar fuera de casa.
- *Visual Studio Code*, como entorno de desarrollo.
- *Slack*, herramienta para comunicarnos con los desarrolladores de Singularity.
- *Github*, servicio online de alojamiento del repositorio de código.
- *Git*, software de gestión de versiones.
- *Overleaf*, editor LaTeX online.
- *Dropbox*, repositorio para documentos.
- *Draw.io*, Herramienta online para realizar diagramas y esquemas.

## 4.5 Valoración de alternativas y plan de acción

Durante el transcurso del proyecto nos podemos encontrar con dos desviaciones. Una es la mala planificación y la segunda es la imposibilidad/viabilidad de integrar CRIU.

### 4.5.1 Planificación incorrecta

En el caso que no hayamos estimado correctamente la planificación debido a que hemos subestimado el esfuerzo necesario se hará un esfuerzo extra en las horas dedicadas al proyecto para que salga adelante. Sabremos si la planificación ha sido correcta mediante nuestro mecanismo de control de gestión, el cual nos proporciona un ratio de eficiencia, que nos permitirá detectarlo.

### 4.5.2 Inviabilidad de integración de CRIU

En este caso sería necesario una inversión de tiempo extra. Ahorraríamos el tiempo necesario para integrar CRIU pero aun así necesitaríamos mas horas ya que la implementación de CRIU, en concreto de *checkpoint/restore*, no es trivial y la integración de CRIU permite un ahorro de tiempo considerable.

Se realizaría la implementación de *checkpoint* de manera nativa en Singularity. Para ello desarrollaríamos una solución inspirada en CRIU pero implementada directamente en Singularity con lo cual nos ahorraríamos los probables problemas de integración de CRIU a costa de un tiempo de desarrollo mayor.

## 4.6 Identificación y estimación de los costes

Las actividades necesarias para llevar a cabo este proyecto tienen diferentes costes que tenemos que tener en cuenta. El presupuesto será desglosado en 4 partes: Costes directos, indirectos, imprevistos y contingencias, además de impuestos.

### 4.6.1 Costes Directos

Estos costes son los que tienen relación directa con la realización del proyecto. En nuestro caso las tareas especificadas en el diagrama de Gantt.

#### 4.6.1.1 Recursos Humanos

Los recursos humanos están divididos en cuatro roles según el tipo de tarea a realizar. Aunque todas las tareas las realizará la misma persona, no es realista asignarles el mismo precio. A cada rol a desempeñar, se le ha sido asignado un precio/hora basado en los salarios pagados en España.[15]

| Tareas | Horas | Recursos | Coste |
|---|---|---|---|
| **1. Planificación inicial** | **70** | | **2.100€** |
| 1.1. Alcance y contexto | 25 | Jefe de Proyecto | 750€ |
| 1.2. Planificación temporal | 10 | Jefe de Proyecto | 300€ |
| 1.3. Presupuesto y sostenibilidad | 15 | Jefe de Proyecto | 450€ |
| 1.4. Documentación final hito inicial | 20 | Jefe de Proyecto | 600€ |
| **2. Análisis del proyecto** | **70** | | **2.100€** |
| 2.1. Estudio implementación Singularity | 25 | Jefe de Proyecto | 750€ |
| 2.2. Estudio implementación CRIU | 15 | Jefe de Proyecto | 450€ |
| 2.3. Estudio implementación CRIU-Docker | 10 | Jefe de Proyecto | 300€ |
| 2.4. Estudio implementación Checkpoint | 20 | Jefe de Proyecto | 600€ |
| **3. Estructuración Singularity** | **85** | | **1.400€** |
| 3.1. Análisis | 15 | Jefe de Proyecto | 450€ |
| 3.2. Diseño | 20 | Diseñador | 400€ |
| 3.3. Implementación | 40 | Programador | 400€ |
| 3.4. Validación | 10 | *Tester* | 150€ |
| **4. Integración CRIU-Singularity** | **120** | | **1.875€** |
| 4.1. Análisis | 20 | Jefe de Proyecto | 600€ |
| 4.2. Diseño | 20 | Diseñador | 400€ |
| 4.3. Implementación | 65 | Programador | 650€ |
| 4.4. Validación | 15 | *Tester* | 225€ |
| **5. Finalización del proyecto** | **50** | | **1.500€** |
| 5.1. Evaluación | 10 | Jefe de Proyecto | 300€ |
| 5.2. Memoria | 20 | Jefe de Proyecto | 600€ |
| 5.3. Presentación | 20 | Jefe de Proyecto | 600€ |
| **Total** | **400** | | **8.975€** |

Tabla 2: Costes directos por actividad.

| Rol | Precio/hora |
|---|---|
| Jefe de Proyecto | 30€ |
| Diseñador | 20€ |
| Programador | 10€ |
| *Tester* | 15€ |

Tabla 3: Precio/hora por rol.

#### 4.6.1.2 Amortizaciones

En esta sección incluimos el precio del uso proporcional del hardware/software durante el transcurso del proyecto. Todo el software que utilizaremos es de uso libre y gratuito, por lo tanto no hay que asumir ningún coste extra por su uso.

El coste de los recursos hardware utilizados viene dado por la siguiente fórmula:

$$\frac{Precio\ de\ compra\ del\ equipo(€)}{Vida\ util\ años\ (4)\ *\ Dias\ laborables\ año\ (220)\ *\ Horas\ de\ trabajo\ diarias\ (8)} *Horas\ de\ uso \quad (1)$$

| Producto | Precio | Horas de uso | Amortización |
|---|---|---|---|
| Ordenador sobremesa | 990€ | 370 | 52€ |
| Ordenador portátil | 820€ | 30 | 4€ |
| Monitor Benq | 130€ | 370 | 7€ |
| Ratón | 80€ | 370 | 4€ |
| **Total** | | | **67€** |

Tabla 4: Amortizaciones Hardware.

#### 4.6.1.3 Costes indirectos

Forman parte de costes indirectos aquellos que no tienen relación directa con las actividades del proyecto pero son necesarios su realización. Estimamos los costes para 4 meses, que es el tiempo dedicado al proyecto.

| Tipo | Coste |
|---|---|
| Luz | 300€ |
| Internet | 240€ |
| **Total** | **540€** |

Tabla 5: Costes indirectos.

### 4.6.2 Imprevistos

Debido a la posibilidad de que surjan imprevistos como en todos los proyectos, destinamos un 5% de los costes directos e indirectos. En nuestro caso sumaría un total de 452€.

### 4.6.3 Contingencias

El mayor riesgo que tenemos es que no sea posible realizar la integración de CRIU. Esto implicaría un retraso aproximado de 60 horas debido a que el tiempo total estimado para la integración de CRIU es 120 horas y la mitad del tiempo aproximadamente se dedicará a CRIU.

Aunque nuestro caso tenga algunas particularidades respecto al resto de tecnologías de contenedores, debido a que han sido capaces de integrar CRIU, asignamos un riesgo de un 15%. Por tanto el coste añadido de la contingencia, que vendrá dado del coste de la fase de integración CRIU-Singularity:

$$1.875€ * 0,5 * 0,15 = 140€ \tag{2}$$

### 4.6.4 Resumen

| **Costes Directos** | **Costes Indirectos** | **Imprevistos** | **Contingencia** | **IVA** | **Total** |
|---|---|---|---|---|---|
| 8.975€ + 67€ | 540€ | 452€ | 140€ | 21% | **12.310€** |

Tabla 6: Resumen presupuesto.

## 4.7 Control de gestión

Debido a la naturaleza del proyecto solo podremos tener una desviación del coste basada en la eficiencia. Los recursos de hardware ya están pagados, la fluctuación de los costes indirectos no tiene un impacto elevado en el proyecto ya que el precio por hora se mantendrá estable y son las horas trabajadas las únicas susceptibles a desviación.

El control de costes a realizar en el proyecto va a consistir en, guardar un registro con las horas y posibles incidencias, por cada tarea, de manera que podremos comprobar si seguimos la planificación establecida y en caso de desviación, seremos capaces de realizar un análisis a las causas de la desviación y el importe extra que supondría para el proyecto.

Analizaremos la eficiencia comparando el número de horas estimadas a cada tarea con las horas dedicadas realmente. Calcularemos el ratio de eficiencia con la siguiente formula:

$$\frac{Horas\ estimadas}{Horas\ reales} \tag{3}$$

Una media de eficiencia menor a 1 a lo largo del proyecto significará que nuestra planificación no ha sido acertada y se habrá utilizado presupuesto de contingencia para corregirlo.

# 5 Planteamiento final

En esta sección vamos a mostrar cual ha sido el planteamiento final del proyecto, justificando las desviaciones y cambios que ha sufrido.

## 5.1 Alcance y Objetivos

Este proyecto finalmente ha consistido en la investigación de las tecnologías Singularity y CRIU, con el objetivo de realizar *Checkpoint/Restore* de aplicaciones simples ejecutándose dentro de un contenedor Singularity. Son aplicaciones que pueden interactuar con ficheros, comunicarse mediante conexiones TCP o que simplemente hacen cálculos y los muestran por pantalla.

Para poder utilizar esta funcionalidad es necesario que el contenedor tenga permisos de root y acceso a CRIU con sus dependencias, además de las dependencias que requiera nuestra aplicación. Esta funcionalidad se lleva a cabo mediante un método sencillo de comunicación entre el anfitrión y una aplicación que se ejecuta dentro del contenedor. Para realizar *Checkpoint/Restore* enviamos el comando necesario a la aplicación que se ejecuta dentro del contenedor y esta se encarga de realizar la acción desde dentro.

### 5.1.1 Obstáculos encontrados

El proyecto inicialmente queria desarrollar una interfaz en Singularity para poder utilizar CRIU. Este desarrollo implicaba implementar una gestión de contenedores en Singularity ya que, como explicamos mas adelante, Singularity no hace ninguna gestión de los contenedores. Esto tiene que ver con el obstáculo que mencionamos al principio del proyecto, que Singularity no dispone de un *daemon* con privilegios de root.

Queríamos haber podido evitar que el contenedor tuviera que ejecutarse con privilegios de root ya que limita los entornos en los que podíamos utilizar esta funcionalidad. Al no poder realizar el *checkpoint* de manera externa, ha sido inevitable.

Habiamos previsto que el mayor obstáculo seria el límite de tiempo y aunque el tiempo ha sido un factor importante, el mayor obstáculo ha sido el hecho de trabajar con tecnologías tan novedosas. El hecho de apenas conocerlas ha hecho que el proyecto haya sido más difícil de lo que pensaba en un primer momento.

## 5.2 Metodología y rigor

Para la realización de este proyecto se ha seguido un método de trabajo iterativo y cíclico. La metodología no ha cambiado respecto al inicio excepto por las fases que hemos seguido y que la metodología de trabajo con Git no se ha seguido, ya que la fase de desarrollo de código no se ha llevado a cabo y no hemos trabajado con Git.

Hemos seguido una metodología iterativa debido al desconocimiento de las tecnologías y tareas que se necesitaban llevar a cabo. De esta manera hemos trabajado según las necesidades que surgían a lo largo del proyecto.

Menos la documentación del proyecto, el resto ha seguido fases de investigación que han llevado a experimentación y desarrollo para luego pasar a fases de evaluación en las que decidíamos cual sería nuestro siguiente paso.

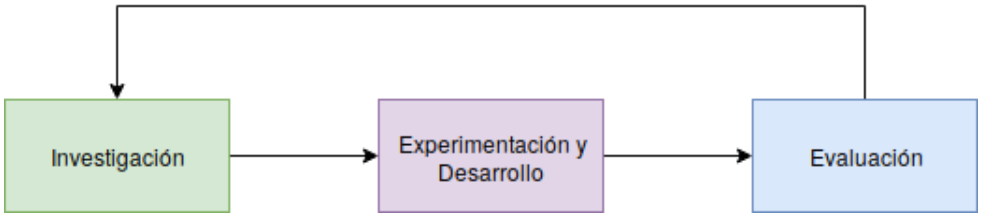


Figura 6: *Método iterativo final.*

### 5.2.1 Método de validación

Para la comprobación de que los resultados obtenidos eran correctos hemos decidido que utilizar aplicaciones con un estado de ejecución sencillo.

El uso de aplicaciones en las que el estado de ejecución avance de una manera predecible, como avanza un contador, y que muestren su estado al exterior constantemente, ya sea escribiendo en la consola o en un fichero, nos permite tener la certeza de que las aplicaciones se han restaurado correctamente, lo que utilizando aplicaciones más complejas podría haber sido más difícil de comprobar.

La comprobación consiste en mirar cual es el último estado que ha comunicado la aplicación y compararlo con el estado que muestra al restaurarse.

## 5.3 Planificación

La planificación del proyecto se ha visto alargada en el tiempo, haciendo que la finalización del mismo se retrasara hasta abril. Esto se ha debido principalmente a que la dedicación de horas semanales prevista no fue posible, debido al trabajo, y decidimos alargar en el tiempo el proyecto para llegar a una solución consistente y bien fundamentada.

La planificación efectiva que ha tenido el proyecto se divide en 4 fases: Documentación inicial, Análisis de las tecnologías, Integración CRIU-Singularity, Finalización del proyecto. La fase de Estructuración de Singularity no se ha llevado a cabo.

### 5.3.1 Revisión: Estructuración de Singularity

Esta fase estaba dedicada al desarrollo de código sobre Singularity para poder integrar CRIU. Esta integración de código no se ha llevado a cabo debido a la necesidad de implementar un sistema de gestión de contenedores en Singularity, el cual tendría que hacer la función de *Checkpoint/Restore*. Esta era una tarea extra que sería compleja de realizar y no era un objetivo del proyecto de manera que, decidimos dejar a un lado el *Checkpoint/Restore* de los contenedores y aplicaciones para centrarnos en el *Checkpoint/Restore* de las aplicaciones que se ejecutan dentro del entorno del contenedor. Para ello nos valdremos de las herramientas proporcionadas por CRIU y Singularity.

### 5.3.2 Documentación inicial

En esta fase se ha realizado una planificación inicial del proyecto. Se contextualizó el proyecto, se formuló el problema, alcance, objetivos, metodologías, planificación de tareas, evaluación de riesgos y costes.

### 5.3.3 Análisis de las tecnologías

Se ha realizado un estudio de Singularity y CRIU principalmente. Como se crean los contenedores Singularity, sus características, diferencias con Docker, usos de CRIU, funcionamiento interno, etc. La investigación de las características de estas tecnologías, como funcionan y su implementación han servido de base para empezar a experimentar.

### 5.3.4 Integración CRIU-Singularity

Esta fase ha sido de investigación y experimentación de lo que se puede y no se puede hacer con CRIU y Singularity, los límites que existen al intentar usar CRIU con contenedores Singularity. Ha sido una fase cíclica: investigación, experimentación y evaluación de resultados. Cada pequeña prueba hacia que tuviéramos que investigar más sobre el funcionamiento de los contenedores, Linux, desarrollo de scripts, Singularity y CRIU, para poder continuar. A partir de estas pruebas vemos casos de uso en los que es posible usar CRIU en contenedores Singularity y se ha encontrado una manera óptima y sencilla para llevarlo a cabo, mediante el uso de scripts y las propias opciones de Singularity y CRIU.

### 5.3.5 Finalización del proyecto

Durante esta fase se ha realizado una evaluación del proyecto sobre las soluciones a las que hemos llegado y su compatibilidad con diferentes tipos de aplicaciones. Después se ha llevado a cabo la redacción de la memoria y la preparación de la defensa del proyecto.

A continuación podemos ver una tabla con las horas dedicadas a cada fase y tarea del proyecto.

| **Tareas** | **Horas** |
|---|---|
| **1. Planificación inicial** | **70** |
| 1.1. Alcance y contexto | 25 |
| 1.2. Planificación temporal | 10 |
| 1.3. Presupuesto y sostenibilidad | 15 |
| 1.4. Documentación final hito inicial | 20 |
| **2. Análisis del proyecto** | **55** |
| 2.1. Estudio Singularity | 30 |
| 2.2. Estudio CRIU | 25 |
| **3. Integración CRIU-Singularity** | **155** |
| 3.1. Investigación | 85 |
| 3.2. Experimentación | 40 |
| 3.3. Desarrollo | 10 |
| 3.4. Evaluación | 25 |
| **4. Finalización del proyecto** | **65** |
| 4.1. Evaluación del proyecto | 10 |
| 4.2. Memoria | 35 |
| 4.3. Presentación | 20 |
| **Total** | **350** |

Tabla 7: Horas por tarea dedicadas.

## 5.4 Recursos

Para llevar a cabo el proyecto nos hemos valido de diferentes recursos. Los recursos humanos han consistido en la persona que realizará el proyecto, la cual ha llevado a cabo todas las tareas descritas.

Los recursos materiales, en nuestro caso *hardware* y *software* son los siguientes:

- Ordenador sobremesa y sus periféricos, con sistema operativo Ubuntu 18.04 LTS.
- Ordenador portátil con Ubuntu 18.04 LTS, de manera que podamos trabajar fuera de casa.
- *Visual Studio Code*, como entorno para estudiar el código de Singularity.
- *Sublime Text*, herramienta usada tanto para el desarrollo de scripts como la edición de texto.
- *Slack*, herramienta para comunicarnos con los desarrolladores de Singularity.
- *Github*, servicio online de alojamiento del repositorio de código.
- *Overleaf*, editor LaTeX online.
- *Dropbox*, repositorio utilizado para el proyecto.
- *Draw.io*, Herramienta online para realizar diagramas y esquemas.

## 5.5 Costes del proyecto

Las actividades y los recursos necesarios han tenido un coste. El presupuesto utilizado está desglosado en 4 partes: Costes directos, indirectos, imprevistos y contingencias, además de impuestos.

### 5.5.1 Costes Directos

Estos costes son los que tienen relación directa con la realización del proyecto. En nuestro caso las tareas especificadas en la planificación

#### 5.5.1.1 Recursos Humanos

Los recursos humanos están divididos en cuatro roles según el tipo de tarea a realizar. Aunque todas las tareas las realizará la misma persona, no es realista asignarles el mismo precio. A cada rol a desempeñar, se le ha sido asignado un precio/hora basado en los salarios pagados en España.[15]

| Tareas | Horas | Recursos | Coste |
|---|---|---|---|
| **1. Planificación inicial** | **70** | | **2.100€** |
| 1.1. Alcance y contexto | 25 | Jefe de Proyecto | 750€ |
| 1.2. Planificación temporal | 10 | Jefe de Proyecto | 300€ |
| 1.3. Presupuesto y sostenibilidad | 15 | Jefe de Proyecto | 450€ |
| 1.4. Documentación final hito inicial | 20 | Jefe de Proyecto | 600€ |
| **2. Análisis de las tecnologías** | **50** | | **1.650€** |
| 2.1. Estudio Singularity | 30 | Jefe de Proyecto | 900€ |
| 2.2. Estudio CRIU | 25 | Jefe de Proyecto | 750€ |
| **3. Integración CRIU-Singularity** | **155** | | **3.425** |
| 3.1. Investigación | 85 | Jefe de Proyecto | 2.550€ |
| 3.2. Experimentación y Desarrollo | 50 | Programador | 500€ |
| 3.3. Evaluación | 25 | QA/Tester | 375€ |
| **4. Finalización del proyecto** | **65** | | **1.950** |
| 4.1. Evaluación del proyecto | 10 | Jefe de Proyecto | 300€ |
| 4.2. Memoria | 35 | Jefe de Proyecto | 1.050€ |
| 4.3. Presentación | 20 | Jefe de Proyecto | 600€ |
| **Total** | **350** | | **9.125€** |

Tabla 8: Costes directos de las actividades realizadas.

| Rol | Precio/hora |
|---|---|
| Jefe de Proyecto | 30€ |
| Programador | 10€ |
| *Tester*/QA | 15€ |

Tabla 9: Precio/hora por rol.

#### 5.5.1.2 Amortizaciones

En esta sección incluimos el precio del uso proporcional del hardware/software durante el transcurso del proyecto. Todo el software que hemos utilizado es de uso libre y gratuito, por lo tanto no hay que asumir ningún coste extra por su uso.

El coste de los recursos hardware utilizados viene dado por la siguiente fórmula:

$$\frac{Precio\ de\ compra\ del\ equipo(€)}{Vida\ util\ años\ (4)\ *\ Dias\ laborables\ año\ (220)\ *\ Horas\ de\ trabajo\ diarias\ (8)} *Horas\ de\ uso \tag{4}$$

| **Producto** | **Precio** | **Horas de uso** | **Amortización** |
|---|---|---|---|
| Ordenador sobremesa | 990€ | 270 | 38€ |
| Ordenador portátil | 820€ | 80 | 9€ |
| Monitor Benq | 130€ | 270 | 5€ |
| Ratón | 80€ | 270 | 3€ |
| **Total** | | | **55€** |

Tabla 10: Amortizaciones Hardware.

### 5.5.2 Costes indirectos

Forman parte de costes indirectos aquellos que no tienen relación directa con las actividades del proyecto pero son necesarios para su realización.

| **Tipo** | **Coste** |
|---|---|
| Luz | 525€ |
| Internet | 420€ |
| **Total** | **945€** |

Tabla 11: Costes indirectos 7 meses.

### 5.5.3 Imprevistos y Contingencias

El incremento del coste debido a imprevistos y contingencias no ha sucedido, ya que no se han superado las horas previstas para el proyecto.

$$1.875€ * 0,5 * 0,15 = 140€ \tag{5}$$

### 5.5.4 Resumen

Podemos ver que el coste de algunas tareas ha aumentado debido a que las horas realizadas por el jefe del proyecto han sido mayores que las estimadas y el coste en general ha disminuido debido a que la cantidad de horas de realización ha sido menor y por tanto vemos como el presupuesto final es similar al previsto.

| **Costes Directos** | **Costes Indirectos** | **Imprevistos** | **Contingencia** | **IVA** | **Total** |
|---|---|---|---|---|---|
| 9.125€ + 55€ | 945€ | - | - | 21% | **12.250€** |

Tabla 12: Resumen presupuesto,

## 5.6 Revisión: Valoración de alternativas, plan de acción y control de gestión

Las desviaciones que pensamos podrían darse a lo largo del proyecto se han acabado produciendo. Vamos a explicar que es lo que ha pasado.

### 5.6.1 Planificación Incorrecta

En un principio planificamos el proyecto como un proyecto de desarrollo de software. Llego un momento en el que vimos que la solución inicial de implementar el soporte a CRIU dentro de Singularity estaba fuera del ámbito del proyecto y aún no conocíamos bien las tecnologías con las que tratábamos, por lo que, como hemos comentado anteriormente, pasamos a intentar realizar el *checkpoint/restore* de las aplicaciones dentro de los contenedores con las herramientas disponibles en CRIU y Singularity.

Hablamos de hacer un esfuerzo extra en las horas dedicadas. Finalmente, estas horas extra de dedicación no se han producido y no hemos llegado a la planificación inicial. Eso se ha debido a una mala gestión del tiempo, al no compaginar correctamente el proyecto con el trabajo fuera de la universidad.

### 5.6.2 Inviabilidad de integración de CRIU

Mencionamos que ante la posibilidad de que no fuera posible integrar CRIU en Singularity se realizaría el desarrollo del mecanismo de *Checkpoint/Restore* de manera nativa, inspirándonos en CRIU, aunque no fuera trivial.

A lo largo del proyecto hemos comprobado que no es trivial y que es una tarea mucho más compleja de lo que pensamos al principio. Subestimamos en gran medida la dificultad de implementación de *Checkpoint/Restore*. Lo verdaderamente inviable era no utilizar CRIU.

# 6 Sostenibilidad del Proyecto

Para evaluar la sostenibilidad del proyecto vamos a estimar el impacto del proyecto dividido en 3 ámbitos diferentes: ambiental, económico y social. Nos basamos en la matriz de sostenibilidad del TFG.[16]

## 6.1 Ambiental

### 6.1.1 Proyecto puesto en producción (PPP)

El impacto sobre el medio ambiente de la realización del proyecto se limita al consumo del equipo que hemos utilizado.

### 6.1.2 Vida útil

Al ser un proyecto software, la huella ecológica que tendrá es totalmente dependiente de las máquinas sobre las que se ejecute y si se hace un uso intensivo de ellas. La solución del proyecto implica una posible mejor utilización de los recursos de computación por lo que las máquinas podrían estar menos tiempo encendidas y gastarían menos electricidad. Pero al mismo tiempo, una optimización en el uso de los recursos de computación implica que las máquinas podrían estar encendidas utilizando su máximo potencial, lo que provoca un incremento del gasto eléctrico.

### 6.1.3 Riesgos

El único riesgo es que aumente la huella ecológica debido a la utilización de todos los recursos de computación. Las tareas se podrán realizar en menos tiempo pero eso no implica que ese tiempo ahorrado no se use para seguir ejecutando nuevas tareas. Por tanto, ante una optimización del número de tareas realizadas por unidad de tiempo la huella ecológica aumentará.

## 6.2 Económico

### 6.2.1 Proyecto puesto en producción (PPP)

Una vez revisados los costes, creo que son unos costes razonables para que el proyecto sea competitivo.El consumo de recursos del proyecto tanto materiales como humanos, con su respectivo coste y desglose por tareas lo podemos encontrar en la sección 5.5.

Para que el coste del proyecto fuera el mínimo posible, todo el software utilizado es gratuito y no se ha tenido que comprar hardware adicional.

Finalmente, el presupuesto planificado no se ha superado. Ha quedado bastante ajustado aún con las diferencias de planificación inicial y final. Ha habido más horas de las previstas ejercidas por el Jefe de Proyecto y eso junto con la diferencia de horas en la planificación, ha balanceado el coste y no hemos tenido que hacer uso del presupuesto para contingencias.

### 6.2.2 Vida útil

El problema que abordamos en este proyecto, *checkpoint/restore* de contenedores, está en proceso de resolución para otras tecnologías de contenedores pero para Singularity no hay nada hecho.

Debido a que es una solución software no tenemos en cuenta algún coste de mantenimiento o ajustes. Actualizaciones futuras deberían ser analizadas.

### 6.2.3 Riesgos

La posible inviabilidad del proyecto viene dada por el hecho de que nuestra solución está limitada a la necesidad de ser un usuario privilegiado para poder utilizarse. En escenarios como HPC está limitación es suficiente para decidir no decantarse por nuestra solución.

## 6.3 Social

### 6.3.1 Proyecto puesto en producción (PPP)

La realización de este proyecto ha supuesto para mí un gran reto. Era muy atractivo el poder trabajar en algo de lo que hay necesidad real, y en tecnologías de las que apenas hemos oído hablar en la carrera.

He tenido la oportunidad de integrar conocimientos que he ido obteniendo durante la carrera, he necesitado asimilar conocimientos completamente nuevos para mi y superar las dificultades que eso conlleva, junto con las propias dificultades de el proyecto. He aprendido a gestionar mejor mi tiempo y como afrontar mejor un proyecto de este tipo. He dado lo mejor de mí y creo que ese esfuerzo a lo largo del proyecto me ha enriquecido como profesional pero sobretodo en el ámbito personal.

### 6.3.2 Vida útil

El impacto de este proyecto ayudará a que los usuarios de Singularity puedan acceder a la funcionalidad de *Checkpoint/Restore* sin tener que cambiar a otra tecnología de contenedores. En algunos casos el cambio de tecnología de contenedores no era posible debido a que necesitaban utilizar las características de Singularity. De esta manera, podrán utilizar mejor los recursos de las máquinas, ser más tolerantes a errores y en general ser más eficientes.

### 6.3.3 Riesgos

Una de las limitaciones de nuestra solución es que los usuarios deben tener privilegios de root dentro del sistema. En entornos HPC no van a disponer de esos privilegios por lo que no van a poder hacer uso de nuestra solución al problema de *checkpoint/restore* en Singularity.

# 7 Leyes y regulaciones relativas al proyecto

Singularity empezó como un producto de código abierto al que cualquiera puede contribuir y acceder. Actualmente, se encuentra bajo la compañía Sylabs, que se fundó debido a la necesidad del mercado de esta tecnología, de manera que ofrecen una versión *Enterprise* de Singularity, que es la misma versión pero más refinada y posiblemente personalizada. Eso si, sin dejar de mantener y desarrollar la versión de código abierto.

En caso de uso de Singularity, modificando o sin modificar el código, se debe proveer el documento relativo a la licencia de Singularity. Además no se deberán utilizar los nombres de los contribuidores o la empresa propietaria para promocionar productos derivados de Singularity, sin el permiso especifico por escrito.

Además hay un *Contributor's Agreement* mediante el cual se especifican las reglas para contribuir al código. El hecho de contribuir implica que, si contribuyes y no impones un acuerdo de licencia escrito separado para tus contribuciones, entonces por la presente otorgas la licencia por defecto del proyecto.

> *A non-exclusive, royalty-free perpetual license to install, use, modify, prepare derivative works, incorporate into other computer software, distribute, and sublicense such enhancements or derivative works thereof, in binary and source code form*

Por lo tanto, nuestro proyecto no se ve limitado por ninguna licencia o regulación.[17]

# 8 Análisis de las tecnologias

En esta sección analizaremos las tecnologías sobre las que vamos a trabajar Singularity y CRIU. Como se utilizan, su funcionamiento y características.

## 8.1 Singularity

Para poder trabajar con Singularity necesitábamos saber como se generan las imágenes, a partir de las cuales se crean los contenedores y, como se crean los contenedores.

### 8.1.1 Imagen de Contenedor Singularity

Uno de los conceptos básicos de diseño de los contenedores Singularity se basa en poder guardar el contenedor como un único fichero *runtime*. Esta decisión permite que el contenedor, que puede tener varios cientos de ficheros internos, sea más fácil de gestionar y además permite funcionalidades interesantes como:

- Rápido acceso a cualquier segmento del contenedor.
- Facilidad para clasificarlo dentro de un sistema con controles de acceso a ficheros.
- Excelente movilidad y reproducibilidad ya que mover o copiar un contenedor es simplemente mover o copiar un fichero.

El formato utilizado por Singularity para las imágenes de sus contenedores es el siguiente.

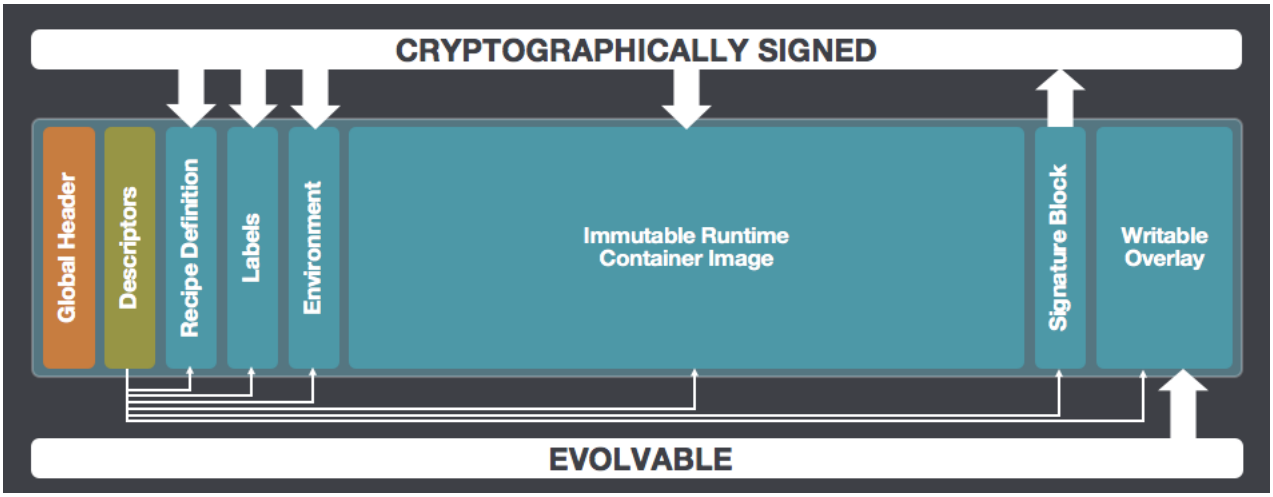


Figura 7: *SIF(Singularity Image Format).*

[18]

### 8.1.2 Generación del SIF

Para generar el SIF necesitamos utilizar el comando **singularity build $imagen $foco** que genera una imagen a partir de un foco de información. El foco define el método que usa Singularity para crear el contenedor y puede ser:

- Imagen ya existente en la Container Library, en el Docker Hub o el Singularity Hub. Todos ellos son repositorios de imágenes de contenedores. Como vemos Singularity puede crear imágenes a partir de Docker.
- Localización de una imagen ya creada en la máquina.
- Localización de un directorio *sandbox*.
- Localización de un *Singularity definition file*.

*Build* produce contenedores en 2 formatos diferentes que pueden convertirse de un formato al otro usando el mismo comando:

- Singularity Image File (SIF) comprimido y con permisos de solo lectura, pensado para ponerlo en producción.
- Directorio (Ch)root (*sandbox*) con permisos de escritura pensado para su uso en el desarrollo de la imagen del contenedor. Opera como un contenedor en un fichero SIF.

Decidimos que lo mejor seria crear contenedores a partir de *Singularity definition files* de manera que tuviéramos un control total sobre los contenedores que utilizaremos.[19]

### 8.1.3 *Singularity definition files*

Este fichero nos permite crear un contenedor completamente personalizado a nuestras necesidades. Incluye especificaciones sobre el Sistema operativo base o el contenedor base de la imagen, además de la instalación de software, configuración de variables de entorno al ejecutar el contenedor, metadatos, etc.

Este fichero está dividido en 2 partes:

- **Cabecera:** Describe el núcleo de sistema operativo que se utilizara dentro del contenedor. Puedes especificar la distribución Linux, la versión, y los paquetes que han de ser parte del núcleo de la instalación.
- **Secciones:** Todas las secciones son opcionales y pueden repetirse dentro del mismo fichero. Diferentes secciones añaden diferentes contenidos o ejecutan comandos.
  - **% setup**, Los comandos de esta sección son ejecutados en el sistema anfitrión, fuera del contenedor, y son ejecutados con privilegios de root, una vez el núcleo del sistema ha sido creado.
  - **%files**, Permite copiar ficheros del sistema anfitrión en el contenedor.
  - **%environment**, Permite definir variables de entorno que se asignarán al ejecutar la imagen del contenedor.
  - **%post**, Comandos definidos son ejecutados en el interior del contenedor después del núcleo del sistema. En esta sección se instalaría software especifico, librerias, se crearian archivos de configuración, etc.
  - **%runscript**, Los contenidos de esta sección se escriben en un fichero dentro del contenedor, que se ejecuta cuando se crea el contenedor con la opción *run*.
  - **%startscript**, Sección similar a la anterior, el contenido se escribe en un fichero dentro del contenedor y este se ejecuta al iniciar una instancia Singularity.
  - **%test**, Esta sección se ejecuta al final para que puedas hacer las pruebas necesarias sobre tu contenedor y comprobar que se creó correctamente.
  - **%labels**, Sección utilizada para añadir metadatos del contenedor.
  - **%help**, Sección en la que añadir la ayuda necesaria para utilizar el contenedor.
  - **%app***, Las secciones que empiezan por app se utilizan para instalar y configurar diferentes aplicaciones en el contenedor que tienen dependencias equivalentes.

Para empezar nuestras investigaciones decidimos utilizar una imagen, trusty.sif, su definition file puede ser consultado en el anexo 12, que simplemente tenia la cabecera definida, basada en ubuntu 14.04.[20]

### 8.1.4 *Namespaces* de Linux

En las siguientes secciones mencionaremos los *namespaces* creados en los contenedores Singularity así que antes vamos a explicar que son, que tipos hay y como funcionan los que son relevantes para el proyecto.

Los *namespaces* son esenciales para el correcto aislamiento de los contenedores. El próposito de los *namespaces* es el de envolver un recurso global del sistema en una abstracción que haga parecer que los procesos dentro del *namespace* tienen una propia y aislada instancia de este recurso. Provee a los procesos con la ilusión de que los recursos que ven son los únicos que existen en el sistema. Por ejemplo, los procesos dentro de un *mount namespace* tienen acceso a una serie de directorios, pero eso no significa que no haya otros *mount namespaces* en los que haya otros directorios además de los suyos. Los procesos quedan aislados de manera totalmente transparente.[21]

Debemos mencionar también los cgroups. Esta es una funcionalidad que permite aislar los recursos que pueden utilizar los procesos, como puede ser la CPU y la memoria. Cgroups y *namespaces* nos permiten aislar lo que los procesos pueden usar y lo que pueden ver. Esto hace que sean herramientas indispensables para los contenedores.[22]

Actualmente hay 6 tipos de *namespaces*:

- *Mount namespace*: Aísla el conjunto de puntos de montaje del sistema de ficheros que un grupo de procesos puede ver.
- *PID namespace*: Aísla el espacio de números identificadores de proceso, PID.
- *UTS namespace*: Aísla dos identificadores de sistema referidos al nombre del anfitrión y nombre del dominio.
- *IPC namespace*: Aísla los recursos de comunicación entre procesos, IPC. Concretamente, objetos IPC de System V y colas de mensajes POSIX.
- *Network namespace*: Aísla los recursos del sistema asociados a las conexiones de red. Como serían las direcciones IP, las tablas de direccionamiento IP, etc.
- *User namespace*: Aísla el los espacios de números identificadores de usuario y grupo.

Vamos a entrar en detalle sobre los *mount namespaces* y los *PID namespaces* que són los que más influencia tienen en el proyecto.

#### 8.1.4.1 *Mount namespaces*

Cada *mount namespace* tiene su propia lista de puntos de montaje, por lo que procesos en diferentes *namespaces* ven y son capaces de manipular diferentes vistas de la jerarquía de directorios común.

Los puntos de montaje pueden cambiarse y eliminarse independientemente. Los cambios en la lista de puntos de montaje son sólo visibles, por defecto, a los procesos pertenecientes al *namespace*. Los cambios son invisbles para los demás *namespaces*.

Cada punto de montaje, tiene un tipo de propagación, que determina si los puntos de montaje creados y eliminados bajo este punto son propagados a otros puntos de montaje. Bajo un mismo *namespace* puede haber puntos de montaje con diferentes tipos de propagación:

- MS_SHARED: Este punto de montaje propaga la eliminación y creación de puntos de montaje, eventos, a los miembros del "*peer group*". Los miembros son los puntos de montaje iguales que existen en varios *namespaces* diferentes o mediante el uso de *mount –bind* que replica el árbol de directorios.
- MS_PRIVATE: Este punto de montaje no propaga ningún evento a su "*peer group*" ni recibe ningún evento.
- MS_SLAVE: Este tipo está entre MS_SHARED y MS_PRIVATE. Un punto de montaje *slave* tiene un *master* que es un "*peer group*" del cual recibe eventos de propagación pero no envia ninguno.
- MS_UNBINDABLE: Este tipo es como MS_PRIVATE y además no puede ser fuente de un *mount –bind*.

[23]

#### 8.1.4.2 *PID namespaces*

Cada *PID namespace* tiene sus propios números de identificación de procesos (PID). Por tanto, procesos en diferentes *namespaces* pueden tener el mismo PID. Esto es útil en la implementación de contenedores que van a ser migrados entre diferentes sistemas así sus PIDs no cambiarán.

Los PID son únicos dentro del *namespace* y asignados a partir del 1. El proceso 1, el proceso *init*, es especial. Es el primer proceso creado dentro del *namespace* y tiene las siguientes características:

- Si un proceso se queda huérfano, se le asignara como padre el proceso *init*.
- Si el proceso *init* termina, el núcleo elimina el resto de procesos en el *namespace*.
- Sólo se le pueden enviar señales, para las cuales el propio *init* haya definido un comportamiento.
- Desde un *PID namespace* padre, se aplica la misma norma con la excepción de las señales SIGKILL y SIGSTOP.

[22]

Cada proceso del sistema Linux tiene un directorio /proc/$PID que contiene información sobre el proceso. Dentro de un *PID namespace* estos directorios muestran información sobre los procesos en el *namespace* o en uno de sus descendientes.

Los *PID namespaces* están anidados jerárquicamente en relaciones padre-hijo. De esta manera desde el *namespace* padre se ve o se es capaz de interactuar con todos los procesos que le corresponden y además los de su hijo. El *namespace* hijo no puede ver los procesos que existen en el padre. Un proceso tendrá un PID en cada una de las capas de la jerarquía de *PID namespaces*, empezando desde abajo.[24]

### 8.1.5 Creación de contenedores

Para crear los contenedores tenemos varios comandos disponibles.

- **singularity shell $imagenDelContenedor**, nos permite ejecutar un terminal en el interior de un contenedor.
- **singularity exec $imagenDelContenedor $comando**, permite ejecutar comandos dentro del contenedor.
- **singularity run $imagenDelContenedor**, ejecuta el script definido en la sección *runscript* del *definition file*. Se consigue el mismo resultado ejecutando la imagen directamente (./imagen.sif). También puede ejecutar aplicaciones definidas en secciones app* **(singularity run –app $nombreApp $imagenDelContenedor)**.

[25]

Estos comandos crean un contenedor en primer plano, a partir de la imagen Singularity especificada. Además, Singularity permite crear contenedores en segundo plano, para ejecutar servicios. Singularity llama a esto instancias.

Las instancias de Singularity surgieron debido a la necesidad de poder ejecutar servicios en segundo plano que no pudieran quedar huérfanos. Al ejecutar un servicio en contenedores creados en primer plano, por ejemplo un servidor web, en el caso de que salieras del contenedor Singularity, el servidor web seguiría corriendo en un inalcanzable *mount namespace*. A este proceso no se le puede eliminar o interactuar con el de manera sencilla, lo que se llama un proceso huérfano, debido a que su padre, el contenedor, ya no esta.[26]

Sobre instancias Singularity se pueden utilizar los comandos descritos anteriormente para ejecutar aplicaciones en el mismo entorno contenido que la instancia.

- Una instancia se ejecuta con la llamada al comando *singularity instance start $imagenDelContenedor $nombreDeLaInstancia* y los comandos anteriores se pueden utilizar sustituyendo la imagen del contenedor por *instance://$nombre de la instancia*. El nombre de la instancia se puede consultar utilizando el comando *singularity instance list*. Este comando lista las instancias que se encuentran en ejecución por el usuario.

### 8.1.6 Características de una imagen en ejecución

Como hemos visto podemos crear contenedores en primer plano o en segundo plano, utilizando instancias. Al crearlos se crea un árbol de procesos y entorno diferente en cada caso:

- Primer plano, Ejecutamos **singularity shell $imagenDelContenedor** y podemos ver que existe un proceso *starter-suid* del que cuelga el *payload* en este caso, *bash*.

  El proceso *starter-suid* crea un proceso sin privilegios que acabara siendo el *payload*.El proceso *starter-suid* trabaja en paralelo con el proceso creado para preparar el entorno en el que va a ejecutarse, el contenedor.

  Aparte de los procesos creados, en este caso, se crea un *mount namespace* para *starter-suid* que actúa de *master* y otro *mount namespace*, como slave, en el que la aplicación se ejecuta. Para comprobar que son *namespaces* diferentes podemos acceder a los *namespaces* de los procesos con el comando *ls -Li /proc/$pid/ns/*.

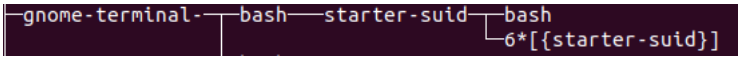


Figura 8: *Árbol de procesos en primer plano.*

- Segundo plano, ejecutamos *singularity instance start $imagenDelContenedor $nombreDeLaInstancia*. Nos encontramos con el mismo caso donde *starter-suid* ayuda a configurar el entorno pero con algunas diferencias. La más clara es que ahora tenemos colgando de *starter-suid* al proceso *sinit* el cual es el padre de los procesos que se ejecuten en el contenedor. Además, como el contenedor se ejecuta en segundo plano, su padre ya no es *bash* sino *systemd*. Para contenedores creados en segundo plano, por defecto se crean los mismos *mount namespaces* para *starter-suid* y otro en este caso para *sinit* y el *payload*. Además se crea un nuevo *PID namespace*, del cual *sinit* es el proceso con PID 1, y un IPC namespace.

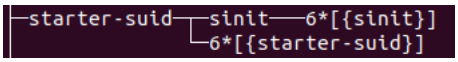


Figura 9: *Árbol de procesos en segundo plano.*

Como hemos dicho, se pueden utilizar los comandos de ejecución en primer plano sobre instancias. Al ejecutarlos sobre la instancia, los procesos que hayamos generado estarán contenidos en el entorno creado previamente para la instancia y formarán parte de los mismos *namespaces*. El árbol de procesos generado es el mismo pero ahora estos procesos están en el *PID namespace* de *sinit* por lo que no pueden quedar huérfanos. Para poder hacerlo sin utilizar la imagen original del contenedor, Singularity guarda información sobre el entorno de la instancia y los espacios de nombres en el sistema anfitrión. Esta información se encuentra en:/var/run/singularity/instances/$usuario/$nombreInstancia

En las siguientes imágenes podemos ver el árbol de procesos y que los procesos pertenecen a los mismos *namespaces*.


Figura 10: *Árbol de procesos en primer plano y segundo plano*

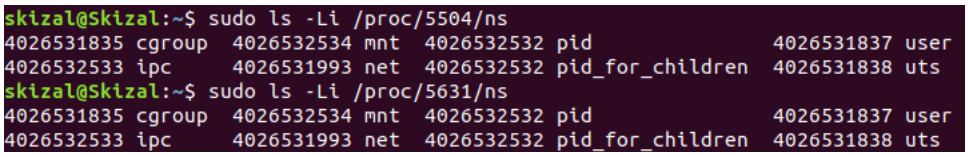


Figura 11: *Namespaces de procesos en primer y segundo plano*

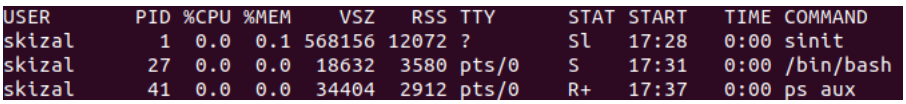


Figura 12: *Procesos visibles desde el proceso bash(5631), en el mismo PID namespace*

La creación de los contenedores en primer o segundo plano no tiene grandes diferencias en cuanto a las tareas que realiza Singularity. Accediendo al código vemos las diferentes funciones que ejecuta Singularity al crear los contenedores.

```
// EngineOperations is an interface describing necessary operations to launch a container process.
type EngineOperations interface {
        // Config returns the current EngineConfig, used to populate the Common struct
        Config() config.EngineConfig
        // InitConfig is responsible for storing the parse config.Common inside
        // the EngineOperations implementation.
        InitConfig(*config.Common)
        // PrepareConfig is called in stage1 to validate and prepare container configuration.
        PrepareConfig(*starter.Config) error
        // CreateContainer is called in master and does mount operations, etc... to
        // set up the container environment for the payload proc.
        CreateContainer(int, net.Conn) error
        // StartProcess is called in stage2 after waiting on RPC server exit. It is
        // responsible for exec'ing the payload proc in the container.
        StartProcess(net.Conn) error
        // PostStartProcess is called in master after successful execution of container process.
        PostStartProcess(int) error
        // MonitorContainer is called in master once the container proc has been spawned. It
        // will typically block until the container proc exists.
        MonitorContainer(int, chan os.Signal) (syscall.WaitStatus, error)
        // CleanupContainer is called in master after the MontiorContainer returns. It is responsible
        // for ensuring that the container has been properly torn down.
        CleanupContainer(error, syscall.WaitStatus) error
}
```

Figura 13: *Funciones para crear un contenedor.*

Estas funciones serán las mismas ya sea un contenedor en primer plano o segundo plano con las pequeñas diferencias que hemos comentado que tienen. Algunos ejemplos:

- La creación de un contenedor en segundo plano implica la creación de un *PID namespace* y la generación de los ficheros que contienen la información sobre el entorno, cosa que en primer plano no se hace.
- La creación de un contenedor en primer plano sobre una instancia, implica que este contenedor tiene que coger la información del entorno de otro lugar y unirse al *PID namespace* de la instancia.

Además, el propio Singularity nos permite ver las etapas por las que pasa un contenedor al ser creado. Al ejecutar imágenes, si utilizamos la opción debug (-d, *singularity -d shell trusty*), se genera una serie de información que nos permite saber que tareas se llevan a cabo y en que función dentro del código. Esta opción nos permite un estudio del flujo de ejecución de la creación de contenedores de una manera más directa. En el anexo 14 podemos encontrar esta información.

### 8.1.7 Comunicación con el contenedor

La comunicación entre el contenedor ya creado y Singularity, no existe. Singularity crea el contenedor mediante el proceso *starter-suid* y este proceso una vez ha creado el entorno del contenedor y tiene el *payload* ejecutándose, se bloquea. Cada contenedor que se crea tiene su propio *starter-suid*, no se comparte.

Un contenedor Singularity, una vez creado, es totalmente independiente con una pequeña excepción, las instancias. Singularity puede acceder al PID del *sinit* de las instancias que están creadas en el sistema, de manera que puede eliminar ese proceso y, en consecuencia, sus hijos.[27]

```go
// MonitorContainer monitors a container
func (engine *EngineOperations) MonitorContainer(pid int, signals chan os.Signal) (syscall.WaitStatus, error) {
        var status syscall.WaitStatus

        for {
                s := <-signals
                switch s {
                case syscall.SIGCHLD:
                        if wpid, err := syscall.Wait4(pid, &status, syscall.WNOHANG, nil); err != nil {
                                return status, fmt.Errorf("error while waiting child: %s", err)
                        } else if wpid != pid {
                                continue
                        }
                        return status, nil
                default:
                        if err := syscall.Kill(pid, s.(syscall.Signal)); err != nil {
                                return status, fmt.Errorf("interrupted by signal %s", s.String())
                        }
                }
        }
}
```

Figura 14: *Función de motorización del contenedor.*

EL proceso *starter-suid* es un proceso bloqueado 14 en un bucle infinito hasta que:

- Los procesos que se están ejecutando terminan, ya sea por error o porque acaben la tarea.
- La instancia es parada con el comando: *singularity instance stop $nombreDeLaInstancia*.

Una vez pasa esto, el proceso se desbloquea y da paso a que el entorno creado anteriormente sea eliminado de manera correcta y no quede rastro del contenedor.[28]

En otras tecnologías como Docker, siempre hay un proceso en ejecución, un *daemon*. Este *daemon* se encarga de poner en marcha las aplicaciones en sus contenedores y gestionarlos. Todos los contenedores Docker son hijos del *daemon* y para poner en marcha un contenedor, has de pedirle al *daemon* que lo haga. Con Singularity pones en marcha la aplicación en su respectivo contenedor sin intermediarios.[29]

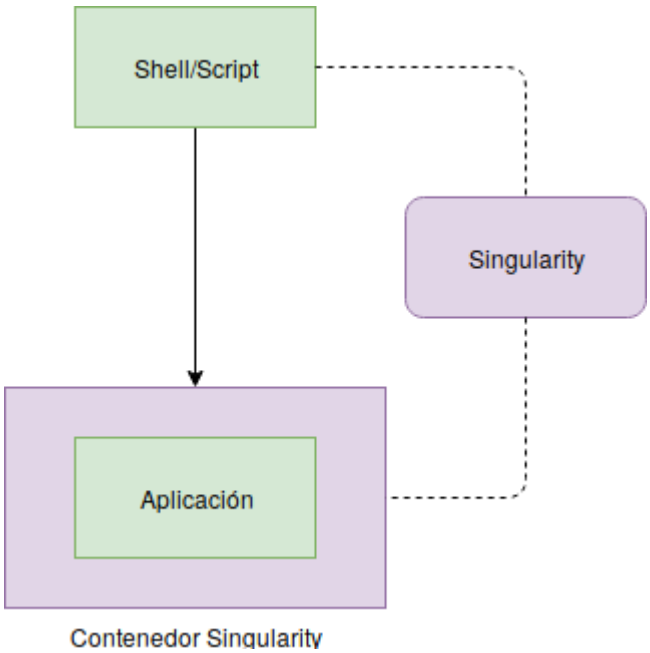


Figura 15: *Singularity creando un contenedor.*

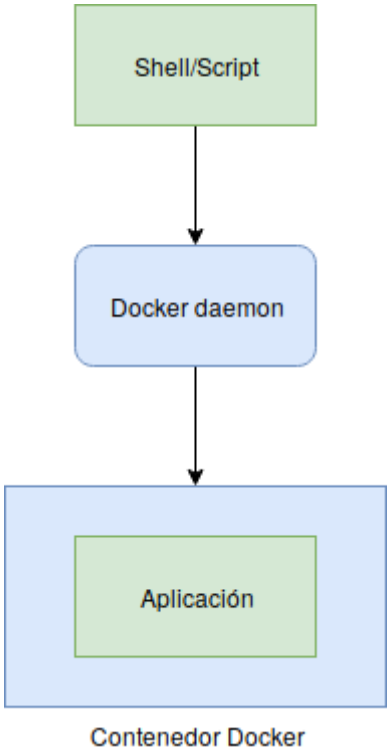


Figura 16: *Docker creando un contenedor.*

## 8.2 CRIU, *Checkpoint/Restore in User Space*

CRIU es una herramienta que nos permite hacer Checkpoint/Restore de aplicaciones de una manera sencilla. Vamos a ver como funciona internamente y como podemos utilizarlo.

### 8.2.1 *Checkpoint*

Para realizar un *checkpoint* necesitamos permisos de root y el comando básico es el siguiente:

*criu dump -t $PID*

El proceso de *checkpoint* depende totalmente del sistema de ficheros /proc. Podemos referirnos a /proc como un pseudo-sistema de ficheros con información de los procesos. No contiene ficheros "reales" sino información del sistema en ejecución, es un centro de control usado por el núcleo del sistema.

De /proc, CRIU recolecta información de los descriptores de ficheros, parámetros de pipes, mapas de memoria, etc.

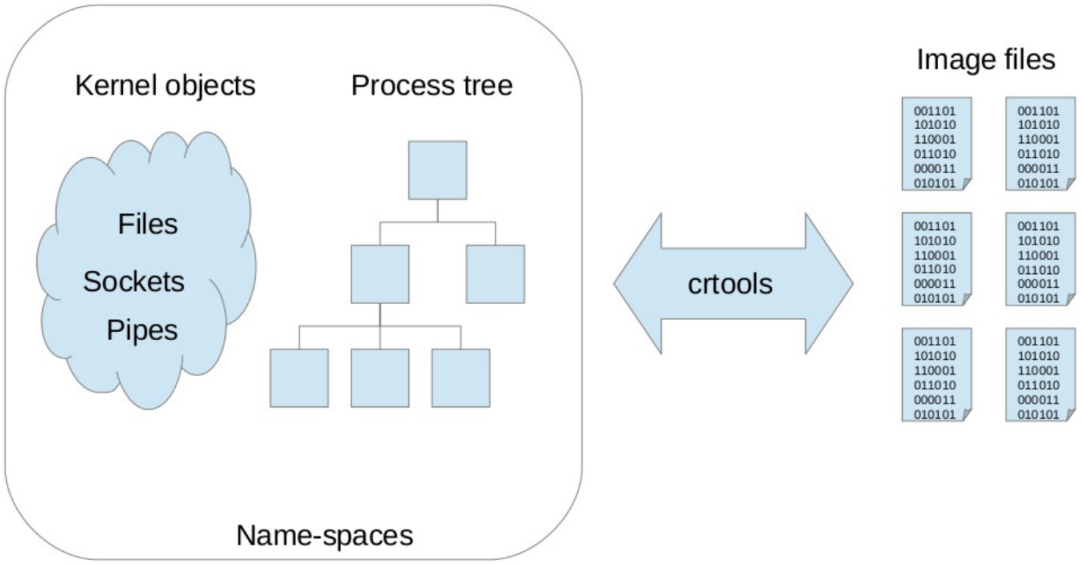


Figura 17: *Funcionamiento CRIU.*

[30]

El proceso que recolecta la información necesaria, *dumper*, lleva a cabo las siguientes tareas:

- **Recopilar el árbol de procesos:** El $PID utilizado en el comando, define el líder del grupo de procesos. Se hará *checkpoint* de todos los hijos. Esto es así para que los procesos hijo no se queden sin padre y porque al restaurar un proceso no puedes asignarlo como padre de otro.

  Entonces, con este $PID el dumper se pasa por el directorio /proc/$PID/task para recopilar threads y sobre /proc/$PID/task/$TID/children para recopilar los hijos del proceso. Las tareas se van parando mientras se recopilan.

  Antes de empezar a hacer *checkpoint* de los procesos, CRIU ha de estar seguro de que no van a cambiar su estado. Cuando decimos cambiar el estado nos referimos a cosas

como abrir nuevos ficheros o sockets, pero sobretodo el hecho de que puedan producir un nuevo proceso hijo. Este proceso hijo podría escapar del proceso de recopilación. Por este motivo, el árbol de procesos ha de ser congelado mientras se recopilan.[31]

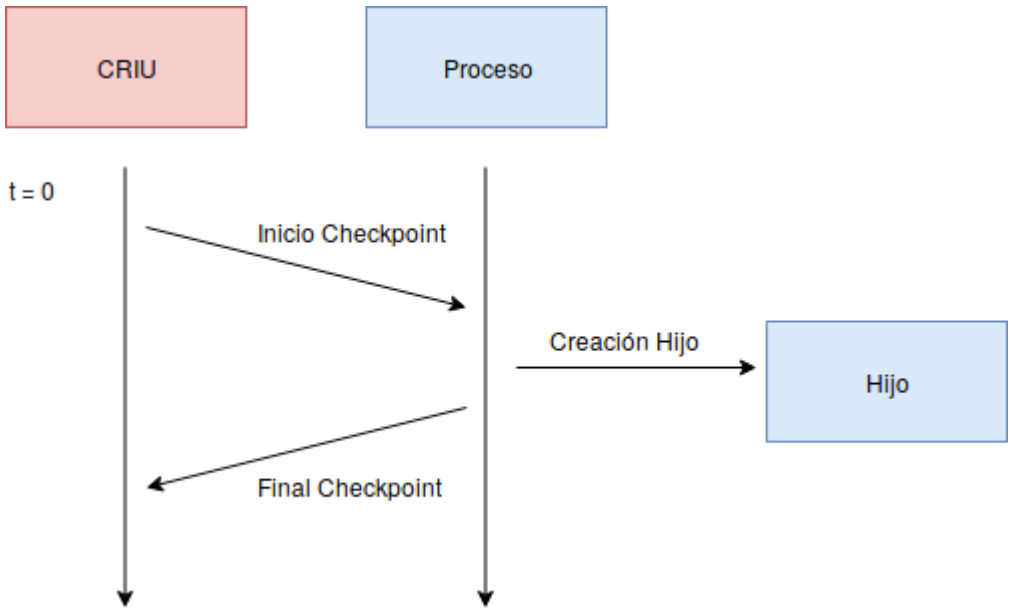


Figura 18: *Escenario a evitar por CRIU.*

- **Congelar árbol de procesos:** El *checkpoint* ha de ser transparente para la aplicación que lo recibe, por tanto no debe ver ningún cambio en la transición del estado del proceso[31]:

  Ejecución $\longrightarrow$ parada $\longrightarrow$ ejecución.

  La manera tradicional de parar y reanudar la ejecución de un proceso es mediante el envió señales de SIGSTOP y SIGCONT. Estas señales son observables desde los procesos que queremos parar y los perturban.

  - SIGSTOP: La señal SIGSTOP no puede ser capturada, bloqueada o ignorada por el proceso pero puede ser vista por procesos padre de espera o *ptracing*.
  - SIGCONT: La señal SIGCONT es especialmente inadecuada ya que puede ser capturada por el proceso.

  Cualquier proceso diseñado para estar pendiente de SIGSTOP y SIGCONT podria romperse al intentar usar estas señales para modificar su estado.[32]

  No se pueden usar estas señales pero hay dos maneras de congelar el árbol de procesos de manera transparente. Capturando las tareas con *ptrace* o utilizando *freezer cgroup*. CRIU utiliza *ptrace*

*Ptrace* es una llamada a sistema que provee un medio por el cual observar o controlar la ejecucion de otro proceso, y poder examinar o cambiar su memoria y registros.

Con PTRACE_SEIZE el proceso es automaticamente parado una vez está conectado con *ptrace* pero sin que el proceso se de cuenta ni cambie su estado. El proceso para con PTRACE_EVENT_STOP y retorna un SIGTRAP en vez de haber enviado SIGSTOP que perturbaría el proceso.

SIGTRAP es una señal que se envía a un proceso cuando una excepción o *trap* ocurre. Un *trap* es una función que define el comportamiento de un proceso al recibir cierta señal. Por ejemplo, cuando utilizamos un debugger, para examinar el flujo de ejecución de un programa, y ponemos un breakpoint para que el programa se interrumpa al ejecutar cierta línea de código, se utiliza SIGTRAP.[33]

- **Recopilar los recursos de las tareas y guardarlos:** En este paso CRIU lee toda la información que conoce sobre las tareas y la escribe en ficheros imagen que se utlizaran luego para restaurar.

  - Las VMA, areas de memoria virtual, del proceso son analizadas desde /proc/$PID/smaps y los ficheros mapeados son leidos desde enlaces en /proc/$PID/map_files
  - Números de descriptores de ficheros son leídos via /proc/$PID/fd.
  - Parámetros centrales de la tarea (como los registros) son volcados mediante la interfaz proporcionada por ptrace y analizados de la entrada /proc/$PID/stat.

  Después de esto, CRIU inyecta un código parásito en la tarea mediante la interfaz ptrace. Este código parásito se utiliza para recibir los descriptores de ficheros, volcar el contenido de la memoria, y escribir la información en diversos ficheros que CRIU puede utilizar para restaurar el proceso.

  El código parasito es un BLOB(Binary Large Object) de código creado en formato PIE (Position-Independent Executable) para la ejecución dentro del espacio de direcciones de memoria de otro proceso. El principal y único cometido de este código es el de ejecutar rutinas de servicio CRIU dentro del espacio de direcciones del proceso del cual se está haciendo *Checkpoint*.

  Cuando se necesita usar el parásito dentro de alguna tarea, procede de la siguiente manera:

  - Se para la tarea mediante el uso de ptrace, PTRACE_SEIZE, para que la tarea no se de cuenta de que alguien intenta manipularla.
  - Inyecta y ejecuta la llamada a sistema mmap dentro del espacio de direcciones del proceso con la ayuda de ptrace, porque se necesita adjudicar un área de memoria compartida que será usada para la cola de memoria del parásito, el intercambio de parámetros entre CRIU y el proceso objetivo, y el socket de control del parásito.
  - Abre una copia local del espacio de memoria compartido de /proc/$PID/map_files, donde PID es del proceso objetivo.

  Cuando las tareas necesarias han sido realizadas y no necesitamos más el parásito, este es eliminado del espacio de direcciones del proceso objetivo:

- CRIU empieza a rastrear las llamadas a sistema que está ejecutando el parásito con la ayuda de ptrace.
- CRIU envia PARASITE_CMD_FINI al parásito mediante el socket de control.
- El parásito recibe el comando, cierra el socket de control y ejecuta la llamada a sistema rt_sigreturn, que restaura la mascara de señales del proceso, cambia las colas y restaura el contexto del proceso objetivo.
- CRIU intercepta el exit de esta llamada a sistema y elimina el mapeado del área de memoria del parásito, así revertiendo el proceso al estado en el que estaba antes de la inyección del parásito.[34]

Después de esto CRIU se desconecta de las tareas y estas, por defecto, mueren. Que por defecto CRIU mate el árbol de procesos tiene sentido. Supongamos que el árbol de procesos sigue ejecutándose, probablemente cambiaran el estado del sistema de ficheros (eliminando un fichero) o el estado de la red (cerrando una conexión TCP). Después de estos cambios CRIU no será capaz de restaurar el árbol de procesos ya que el sistema no tendrá los recursos que este árbol de procesos requiere.

Si el proceso no modifica el estado del sistema, se puede mantener en ejecución mediante la opción –leave-running.

### 8.2.2 *Restore*

El proceso de *Restore*, restorer, se lleva a cabo haciendo que CRIU se transforme en la tarea que restaura. Consiste en 4 fases.

- **Resolver recursos compartidos:** En este paso CRIU lee los ficheros imágenes y descubre que procesos comparten que recursos. Después recursos compartidos son restaurados a el proceso uno y el resto de procesos, o heredan de ese proceso uno en el segundo paso o los obtienen de otra manera. Esto último es, por ejemplo, areas de memoria compartida que se restauran mediante el descriptor de fichero memfd.

- ***Fork* del árbol de procesos:** CRIU realiza la llamada a sistema fork() las veces necesarias para recrear los procesos que han de ser restaurados.

- **Restaurar recursos básicos de las tareas:** CRIU restaura todos los recursos excepto:
  - Localización exacta de los mapeados de memoria.
  - *Timers*.
  - Credenciales
  - *Threads*, hilos de ejecución.

  En esta fase CRIU, abre ficheros, prepara *namespace*, mapea áreas de memoria privada y las llena de datos, crea sockets, etc.

- ***Restorer Context*:** Esta es la ultima fase del proceso *restore*. Difiere del contexto del proceso CRIU como hacia el código parásito en el *checkpoint*. En este contexto CRIU restaura los recursos que quedaban.

Esta fase se hace en un contexto separado ya que, cuando CRIU llega al momento en el que necesita restaurar la memoria del proceso. Debe eliminar un mapeado y mapear otro. Pero como CRIU haría esta operación sobre si mismo, en el momento que el código de CRIU fuera eliminado del mapeado de memoria, se provocara un error fatal en CRIU (segmentation-fault). Se necesita un código que haga eso y ese código deberia estar en 2 espacios de direcciones simultaneamente, el de CRIU y el del proceso.

CRIU aquí solo realiza dos acciones. Mover VMAs anónimas a su posición y mapear nuevas VMAs de ficheros.

Los timers son restaurados aquí, ya que los procesos CRIU pueden esperar los unos a los otros durante algún tiempo mientras restauran sin perder ticks de reloj.

Se restauran las credenciales aquí para permitir a CRIU realizar operaciones privilegiadas como fork-with-pid o chroot().

Threads son restaurados aquí por simplicidad. Si se restauran antes, se deberían dejara un lado mientras se cambia la disposición de la memoria.

Por defecto, los procesos restaurados tienen como padre un proceso criu y este tiene como padre al proceso que haya ejecutado CRIU. Se pueden usar 3 diferentes flags para modificar esto[35]:

- –restore-detached. Hace que el proceso criu termine después de restaurar el árbol de procesos. Esto deja huérfano al proceso restaurado y pasa a ser hijo del proceso init.
- –restore-sibling. Hace que el proceso criu cree otro proceso con privilegios de root, que sea hijo del proceso que ha llamado CRIU. Después del restore el proceso criu termina y el proceso restaurado tiene como padre el proceso que ejecutó CRIU.
- –exec-cmd. Hace que el proceso criu sea reemplazado por otro y este otro será el padre del proceso restaurado.

### 8.2.3 Uso de CRIU

Los comandos básicos para utilizar CRIU són criu dump y criu restore. Vamos a ver algunos flags que utilizan y nos serán útiles:

- **Dump**.
  - *-t $PID*. Este flag es el único obligatorio, es el que indica a que árbol de procesos se hace *checkpoint*.
  - *–leave-running*. Indica que el proceso del que se ha hecho *checkpoint* no debe terminarse.
- **Restore**.
  - *–root $directorio*. Este flag indica que directorio utilizará CRIU como raíz en el caso de crear un *mount namespace*.
  - *–restore-detached (-d) y –restore-sibling*. Para modificar el padre del árbol de procesos restaurado, como comentamos en la sección anterior.
- **Comunes**.
  - *-o $archivoLog*. Indica que archivo se va a generar sobre el proceso de *checkpoint*.
  - *-v hasta -v4*. Indica la cantidad de información que habrá en los log.
  - *–external mnt[KEY]:VAL*. Indica de donde vienen los directorios creados con mount –bind.
  - *–images-dir $directorio*. Indica el directorio donde se guardaran las imágenes generadas por CRIU.
  - *–tcp–established*. Indica que se han de mantener el estado de las conexiones TCP.
  - *–shell-job*. Indica que los procesos comparten sesión con un terminal.

# 9 Integración CRIU-Singularity

En esta sección vamos a hablar de las pruebas realizadas para conocer los límites actuales al usar CRIU con los contenedores Singularity, de manera que sepamos como podemos sacar el máximo provecho a la combinación de estas tecnologías.

Para las pruebas realizadas hemos utilizado la versión 3.0 de Singularity y la versión 3.11 de CRIU. Además, diferentes *Singularity definition files*, que podemos encontrar en el anexo12, de contenedores junto con scripts que nos permiten mostrar variedad de casos de uso.

Los contenedores que hemos usado son:

- Contenedor creado a partir de una imagen original de ubuntu 14.04 (trusty.sif) sin ningún añadido.
- Contenedor creado a partir de una imagen original de ubuntu 18.04 (bionicCRIU.sif) con CRIU instalado.

Hemos utilizado programas simples, que podemos encontrar en el anexo 13, para probar el funcionamiento de CRIU:

- **Contador**. Una manera sencilla de comprobar que el checkpoint/restore funciona correctamente. Consiste en un bucle infinito mediante el cual escribimos por pantalla números sucesivos a una frecuencia fija.
- **Contadores**. Igual que contador pero a frecuencias diferentes. De manera que podemos comprobar que el *checkpoint/restore* funciona correctamente en una jerarquía de procesos.
- **Lector/Escritor**. El Lector lee líneas de un fichero y las muestra por pantalla a una frecuencia fija. El Escritor es como el Contador pero escribiendo los números en un fichero. De esta manera comprobamos que el puntero de lectura/escritura se mantiene y la interacción con ficheros no se ve afectada.
- **Cliente/Servidor TCP contador**. Un servidor que envía números sucesivos a las conexiones que recibe.Para comprobar que la conexión con sockets puede ser restaurada.

## 9.1 *External Checkpoint/Restore*

Para empezar a probar CRIU con Singularity decidimos intentar realizar el Checkpoint/Restore desde fuera del contenedor Singularity. Hacerlo de manera externa era una solución ideal ya que no hacia falta que el contenedor tuviera permisos de root, ni que tuviera acceso a CRIU, solo se necesitaban permisos de root desde el sistema anfitrión que tenia que ejecutar CRIU.

Para llevar a cabo las pruebas utilizamos:

- SIF de contenedor con la versión trusty de ubuntu.
- Script contador de números para apreciar que el estado se restaura en el momento adecuado.

### 9.1.1 *Checkpoint*

Como primer paso decidimos intentar hacer checkpoint del proceso como si de un proceso normal se tratara para ver como reaccionaba CRIU y que tipo de errores daba:

- Ejecutamos un shell dentro del contenedor trusty (singularity -d shell trusty.sif).
- Desde este shell ejecutamos nuestro script contador.
- Desde otro shell ejecutamos ( pstree -p — grep contador ) para ver el PID del proceso contador.
- Una vez sabemos el PID ejecutamos ( criu dump -o dump.log -v4 -t PID –shell-job ), para hacer *checkpoint* del contador.

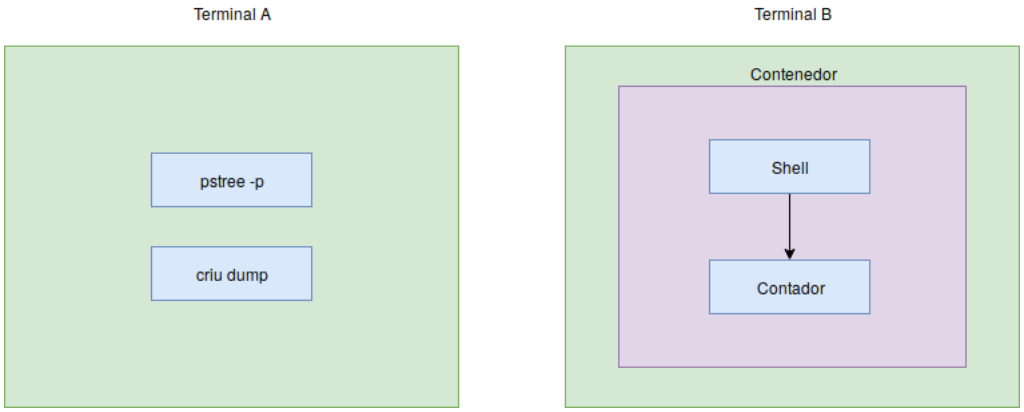


Figura 19: *Comunicación mediante pipe.*

Este último paso no mata el proceso y al mirar el fichero dump.log generado vemos el siguiente error:

```
mnt: Inspecting sharing on 1271 shared_id 0 master_id 761 (@./etc/group)
Error (criu/mount.c:947): mnt: Mount 1271 ./etc/group (master_id: 761 shared_id: 0) has unreachable sharing. Try --enable-external-masters.
Unlock network
Unfreezing tasks into 1
    Unseizing 22003 into 1
Error (criu/cr-dump.c:1748): Dumping FAILED.
```

Figura 20: *Error external checkpoint.*

Este error se debe a que hay diferentes directorios los cuales Singularity recrea dentro del contenedor y CRIU detecta que es un directorio recreado como Singularity crea un nuevo *mount namespace*, no lo ve. Por tanto CRIU necesita guardar estas localizaciones para que al hacer *restore* le indiquemos el directorio original.

Podemos saber que directorios hemos de indicarle de dos maneras:

- Buscando en el fichero debug Singularity (generado con la opción -d) que directorios se han recreado en el interior del contenedor.
- Ejecutar ciclos de prueba y error. A cada error generado por CRIU, vemos el directorio que no está encontrando y se lo indicamos.

El error nos dice que utilicemos –enable-external-masters pero eso nos generaba el mismo error ( es una función que esta en proceso de ser reemplazada y puede ser la causa ). Nos redireccionan a que utilicemos (–external mnt[]:flags) para un proceso automático, pero no cambia nada.

El proceso para indicar los directorios fue manual, utilizando (–external mnt[KEY]:VAL) de manera que al hacer *checkpoint* asignamos una clave a cada directorio y al hacer restore la clave a utilizar es el valor usado anteriormente, mientras que el nuevo valor es el directorio que CRIU tiene que recrear.

- criu dump -o dump.log -v4 -t PID $localizaciónImagenesCheckpoint –shell-job –external mnt[/etc/group]:/etc/group ...

Al añadir todos los directorios el proceso no generaba ningún error extra al hacer *Checkpoint*.


Figura 21: *External Checkpoint con éxito.*

### 9.1.2 *Restore*

Una vez tenemos el *checkpoint* listo, pasamos al *restore*. Utilizamos el comando de restore en el que no hay que indicar ya el PID del proceso. Hay que indicar que directorio estamos recreando y cual es el directorio original.

- criu restore -o restore.log -v4 –images-dir $localizaciónImagenesCheckpoint -d –shell-job –external mnt[/etc/group]:/etc/group ...

Este comando no restaura el proceso por lo que comprobamos el restore.log generado.

```
 22003: Error (criu/mount.c:2499): mnt: The --root option is required to restore a mount namespace
 22003: mnt: Start with 0:/tmp/.criu.mntns.G01MgT
Error (criu/cr-restore.c:1418): 22003 killed by signal 9: Killed
Error (criu/cr-restore.c:2294): Restoring FAILED.
```

Figura 22: *Error external restore.*

Este error nos dice que dice que necesitamos especificar un directorio raíz para restaurar un *mount namespace*, del que va a colgar la jerarquía de directorios relativa al proceso contador por lo que miramos el debug de Singularity.

```
DEBUG   [U=1000,P=21817]   create()      Chroot into /usr/local/var/singularity/mnt/session/final
DEBUG   [U=0,P=21835]      Chroot()      Change current directory to /usr/local/var/singularity/mnt/session/final
```

Figura 23: *Directorio root de Singularity.*

De aquí suponemos que el directorio /usr/local/var/singularity/mnt/session/final es el que tenemos que indicar como root.

- criu restore -o restore.log -v4 –images-dir $localizaciónImagenesCheckpoint -d –shell-job –root /usr/local/var/singularity/mnt/session/final –external mnt[/etc/group]:/etc/group ...

Al ejecutar este comando, no logramos restaurar el proceso y el sistema anfitrión al completo comienza a dar errores como que al abrir un terminal no podemos interactuar con él y nos vemos obligados a reiniciar el sistema.

Pensamos que CRIU crearía directamente el *namespace* y no sabíamos que pasaba. Decidimos probar a cambiar la localización donde CRIU recreara los directorios y encontramos un error distinto.

```
(00.068846) Error (criu/mount.c:3270): mnt: Can't remove the directory /tmp/.criu.mntns.v0tAIg: Device or resource busy
(00.068873) Error (criu/cr-restore.c:2294): Restoring FAILED.
```

Figura 24: *Error eliminación directorio.*

CRIU restaura la jerarquía de directorios bajo un montaje con un fichero temporal como root. Si hay algún error es posible que este directorio quede montado y en subsecuentes llamadas

a CRIU aparezca este error. Para librarnos de el teniamos que, desmontar el directorio mediante la llamada *umount* y eliminarlo.

Ejecutamos de nuevo y encontramos otro error, no encuentra el directorio /etc/localtime. Obviamente, el directorio /etc/localtime existe en nuestra máquina y finalmente decidimos abrir un caso a los desarrolladores de CRIU en su GitHub para ver si nos podían ayudar.

Gracias a ellos vimos que CRIU necesita que los directorios que estaban montados dentro del contenedor, existan de manera idéntica fuera de él. Esos directorios son los que ya probamos anteriormente y dejaron nuestro sistema inutilizado. Para que pudiera funcionar decidimos recrear el sistema de directorios bajo /usr/local/var/singularity/mnt/session/final.

Una vez recreados los directorios encontramos nuevos errores, estos relativos a la aplicación que corría en el contenedor y algunas librerías necesarias. Todo lo que estuviera relacionado con el contenedor debía existir de manera idéntica, incluyendo el tamaño de ejecutables del bin y librerias, los permisos de los ficheros y el tamaño debían ser los mismos, los /dev también. Para conseguir los archivos idénticos descargamos los ficheros originales de ubuntu 14.04 desde el mismo sitio que Singularity utilizo para crear la imagen SIF.

Una vez tuvimos los directorios recreados y con los archivos que necesitaba CRIU el contador se restauró con éxito desde el estado en el que se hizo *checkpoint*.

```
skizal@skizalpc:~$ sudo criu restore -o restore.log -v4 -d --root /usr/local/var/singularity/mnt/session/final --external mnt[/etc/group]:/usr/local/var/sing
ularity/mnt/session/final/etc/group --external mnt[/etc/passwd]:/usr/local/var/singularity/mnt/session/final/etc/passwd --external mnt[/etc/resolv.conf]:/usr
/local/var/singularity/mnt/session/final/etc/resolv.conf --external mnt[/var/tmp]:/usr/local/var/singularity/mnt/session/final/var/tmp --external mnt[/tmp]:/
usr/local/var/singularity/mnt/session/final/tmp --external mnt[/home/skizal]:/usr/local/var/singularity/mnt/session/final/home/skizal --external mnt[/proc/sy
s/fs/binfmt_misc]:/usr/local/var/singularity/mnt/session/final/proc/sys/fs/binfmt_misc --external mnt[/proc]:/usr/local/var/singularity/mnt/session/final/pro
c --external mnt[/.singularity.d/actions]:/usr/local/var/singularity/mnt/session/final/.singularity.d/actions --external mnt[/etc/hosts]:/usr/local/var/singu
larity/mnt/session/final/etc/hosts --external mnt[/etc/localtime]:/usr/local/var/singularity/mnt/session/final/etc/localtime --external mnt[/dev/mqueue]:/usr
/local/var/singularity/mnt/session/final/dev/mqueue --external mnt[/dev/hugepages]:/usr/local/var/singularity/mnt/session/final/dev/hugepages --external mnt[
/dev/shm]:/usr/local/var/singularity/mnt/session/final/dev/shm --external mnt[/dev/pts]:/usr/local/var/singularity/mnt/session/final/dev/pts --external mnt[/
dev]:/usr/local/var/singularity/mnt/session/final/dev --shell-job
skizal@skizalpc:~$ Contador 10
Contador 11
Contador 12
```

Figura 25: *External Restore con éxito.*

Al mirar la jerarquía de directorios, vemos que el proceso no tiene de padre a *starter-suid* como debería. Sino que el padre es systemd, ya que el bash que lo ejecuta no puede ser padre debido a limitaciones de CRIU.

```
├─systemd(1326)─┬─(sd-pam)(1327)
│               ├─at-spi-bus-laun(1440)─┬─dbus-daemon(1445)
│               │                       ├─{at-spi-bus-laun}(1441)
│               │                       ├─{at-spi-bus-laun}(1442)
│               │                       └─{at-spi-bus-laun}(1444)
│               ├─at-spi2-registr(1448)─┬─{at-spi2-registr}(1449)
│               │                       └─{at-spi2-registr}(1450)
│               ├─contador.sh(2445)
```

Figura 26: *Proceso contador tiene de padre a systemd, el proceso init.*

Las opciones de CRIU no nos permiten restaurar el proceso bajo su padre original, o lo que es lo mismo, no nos permiten restaurar el proceso dentro del contenedor en el que fue creado.

Visto el resultado, lo incómodo y artificial de realizar y, que el proceso no se restaura dentro del contenedor con *external checkpoint/restore*, decidimos investigar la posibilidad de realizar la tarea desde dentro del contenedor.

## 9.2 *Internal Checkpoint/Restore*

Al hacer el *Checkpoint/Restore* de manera interna, el contenedor ha de ser ejecutado con permisos de root y debe tener CRIU instalado junto a sus dependencias o recibir las dependencias necesarias junto con CRIU utilizando la opción –bind, que permite recrear directorios específicos dentro del contenedor.

Para llevar a cabo las pruebas utilizamos:

- SIF de contenedor con la versión bionic de ubuntu, con CRIU y dependencias instaladas, definition file se puede encontrar en el anexo12. Para descargar CRIU y dependencias en el contenedor es necesario añadir el repositorio de paquetes en el que se encuentra CRIU.
- Los scripts contador, contadores, escritor, lector y Cliente/Servidor TCP contador. Códigos de scripts en el anexo 13 y código TCP aquí [36]

El primer problema que nos encontramos es que desde Singularity no podemos comunicarnos con el contenedor y necesitamos que se ejecuten los comandos de CRIU dentro de él. Además, al necesitar ejecutar CRIU desde dentro del contenedor, el contenedor ha de crearse con permisos de root. Para realizar el *checkpoint* desde dentro del contenedor debemos ejecutar el contenedor con el comando shell, así podemos trabajar desde dentro.

Los pasos a seguir son:

- Ejecutamos un shell dentro del contenedor bionicCRIU (sudo singularity shell bionic-CRIU.sif)
- Ejecutamos el programa escritor, para poder usar el shell sin que se nos llene de ruido del fichero contador.
- Desde el shell del contenedor ejecutamos (criu dump -o dump.log -v4 -t PID –shell-job), para realizar el *checkpoint* del escritor.
- Abrimos el fichero outputEscritor.txt para ver el último estado y ejecutamos (criu restore -o restore.log -v4 –shell-job)

El restore ha hecho su función y el escritor no ha perdido su estado, lo único que no tenemos control del shell ya que hemos ejecutado el restore en primer plano. Hacemos Ctrl-C para que se elimine el proceso y probamos a ejecutar el comando *restore* en segundo plano utilizando & al final del comando. El proceso se restaura bien y sigue haciendo su trabajo. Probamos a utilizar el shell y el contenedor falla y hace *exit*.

```
[sudo] password for skizal:
root@Skizal:/home/skizal/tfg/memo/test# ./escritor.sh &
[1] 23316
root@Skizal:/home/skizal/tfg/memo/test# criu dump -o dump.log -t 23316 --shell-job
Warn  (criu/kerndat.c:659): Can't load /run/criu.kdat
[1]+  Killed                  ./escritor.sh
root@Skizal:/home/skizal/tfg/memo/test# criu restore -o restore.log -v4 --shell-job &
[1] 23357
root@Skizal:/home/skizal/tfg/memo/test# Warn  (criu/kerndat.c:659): Can't load /run/criu.kdat

exit
```

Figura 27: *Error del contenedor.*

El *warning* que vemos no influye en el error ya que */run/criu.kdat* lo utiliza CRIU como cache, para acelerar el proceso nada más.

Esta solución nos permite hacer todos los *checkpoint* que queramos (si usamos –leave-running el proceso no muere al hacer *checkpoint*) pero una vez hemos restaurado el proceso ya no podemos hacer nada más.

Esto implica que una vez restauremos el proceso, no podremos seguir haciendo *checkpoint* del proceso restaurado. Cualquier intento de interactuar con el shell provoca una salida del contenedor. Además, esta salida del contenedor implica que se quedan huérfanos los procesos restaurados.

### 9.2.1 Solución

Ante este problema de solo poder hacer un restore necesitábamos poder saltarnos esa limitación en la que perdemos el control del shell y falla el contenedor.

Se nos ocurrió una solución simple pero que servia a nuestro propósito, que los comandos de *dump/restore* los ejecutara otro proceso que no fuera el shell. Para que este proceso supiera cuando tenia que ejecutarlos y que comando concreto tenia que ejecutar, decidimos que nos comunicariamos con el proceso utilizando una *pipe* con nombre.

Este proceso, manager.sh, se encargara de hacer *checkpoint/restore* de los procesos que le digamos. El funcionamiento es el siguiente:

- Una vez se ejecute el contenedor, se pone en marcha el manager en segundo plano, usamos & al ejecutar desde la shell.
- El manager al iniciarse crea una *pipe* que está en un directorio que montado con la opción –bind, de esta manera la pipe se ve desde el anfitrión y desde dentro del contenedor. Después entra en un bucle infinito esperando que le lleguen órdenes por la *pipe*.
- Desde un shell enviamos el comando que queremos ejecutar a través de la pipe.
- El manager lee el comando y lo ejecuta.

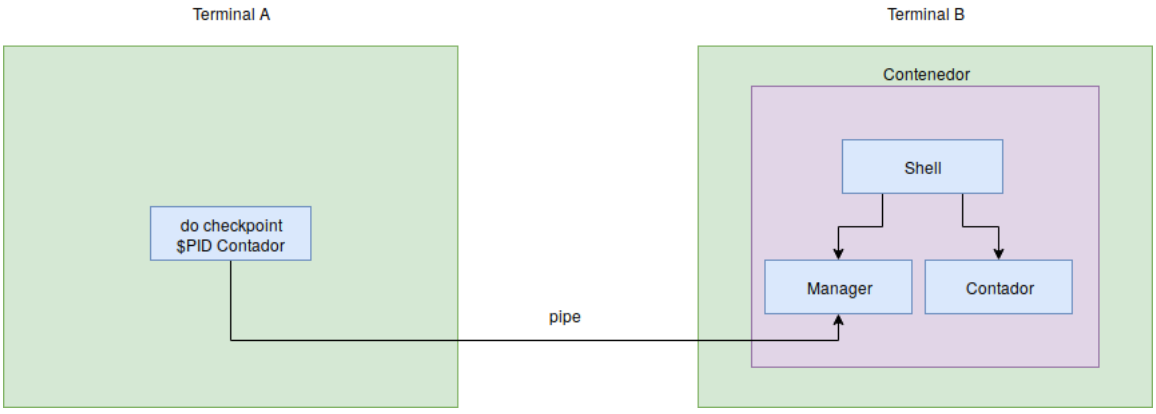


Figura 28: *Comunicación mediante pipe.*

De esta manera el shell tiene el mismo problema por el cual una interacción con el hace que falle pero, como enviamos los comandos al manager y el los ejecuta, no el shell, podemos

hacer *checkpoint/restore* tantas veces como necesitemos o queramos. Como es el manager el que ejecuta los comandos criu, cuando restauremos los procesos, estos serán hijos suyos.

Las imágenes que genera CRIU al hacer *checkpoint* deben guardarse en una localización que también haya sido montada con la opción –bind. De esta manera podemos hacer *checkpoint* de un proceso ejecutándose en un contenedor y restaurarlo en otro contenedor que se haya creado a partir de la misma imagen.

Además podemos evitar el problema de procesos huérfanos al salir del contenedor de una manera sencilla. Le ordenamos al manager que haga *checkpoint* de los procesos y luego le decimos que finalice el bucle infinito para que el mismo tampoco se quede huérfano.

En el caso de las instancias hace falta que el manager este ejecutandose dentro del *PID namespace* y que la aplicación se haya ejecutado utilizando el comando setsid delante debido a este error.

```
(00.004830) sid=0 pgid=0 pid=48
(00.004832) Error (criu/cr-dump.c:1335): A session leader of 48(48) is outside of its pid namespace
```

Figura 29: *Error Checkpoint en PID namespace.*

### 9.2.2 Pruebas de *checkpoint/restore* con el manager

Una vez que hemos encontrado una manera de poder realizar *checkpoint/restore* pasamos a hacer pruebas con los scripts mencionados anteriormente, para ver que todo funciona bien.

Para que el uso del manager sea más accesible hemos desarrollado los scripts checkpoint.sh y restore.sh. Estos scripts como argumentos necesitan:

- Localización donde ha creado la pipe el manager.
- Localización donde ha de guardar las imágenes criu.
- PID del proceso al que queremos hacer *checkpoint* (solo para checkpoint.sh)
- Tipo de *checkpoint*, hay 5 opciones:
  - 0: –shell-job
  - 1: –tcp-established
  - 2: –shell-job –tcp–established
  - 3: –shell-job –leave-running
  - 4: –shell-job y quit. Para evitar procesos huérfanos, hace *checkpoint* de los procesos y termina el manager.

Para todas las pruebas vamos a mostrar el árbol de procesos y el estado del proceso del cual hacemos *checkpoint*, antes y después.

Hemos utilizado la imagen bionicCRIU.sif, que podemos encontrar en el anexo 12, para hacer las pruebas.

#### 9.2.2.1 Contador

En un terminal hemos ejecutado:

- sudo singularity shell bionicCRIU.sif
- ./manager.sh $pipe &
- ./contador.sh

El resto de comandos se ejecutan desde otro terminal con permisos root.

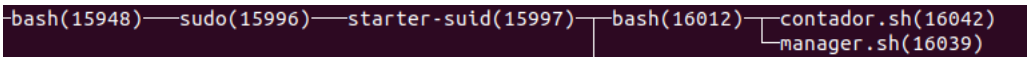


Figura 30: *Árbol de procesos contador y manager.*

Ejecutamos: ./checkpoint.sh $pipe $directorioImagenes $PIDContador 0

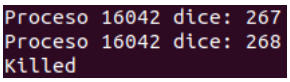


Figura 31: *Estado del contador antes del checkpoint.*

Ejecutamos: ./restore.sh $pipe $directorioImagenes 0

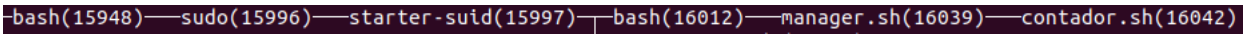


Figura 32: *Árbol de procesos manager y contador al restaurar.*

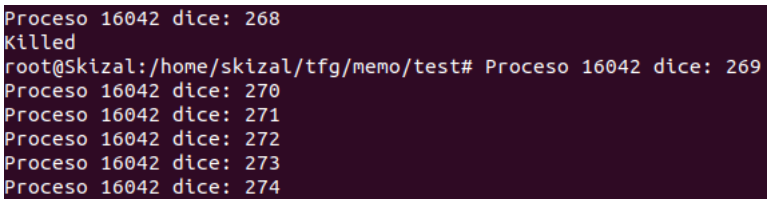


Figura 33: *Estado del contador al restaurar.*

#### 9.2.2.2 Contadores

En un terminal hemos ejecutado:

- sudo singularity shell bionicCRIU.sif
- ./manager.sh $pipe &
- ./contadores.sh

El resto de comandos se ejecutan desde otro terminal con permisos de root.

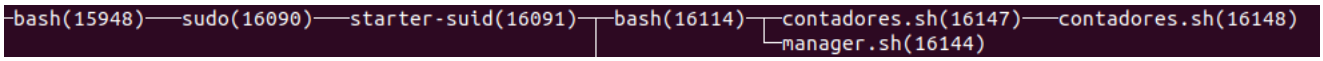


Figura 34: *Árbol de procesos contadores y manager.*

Ejecutamos: ./checkpoint.sh $pipe $directorioImagenes $PIDContadoresPadre 0

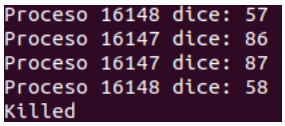


Figura 35: *Estado de contadores antes del checkpoint.*

Ejecutamos: ./restore.sh $pipe $directorioImagenes 0

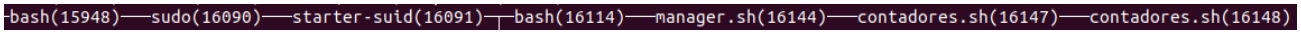


Figura 36: *Árbol de procesos contadores y manager al restaurar.*

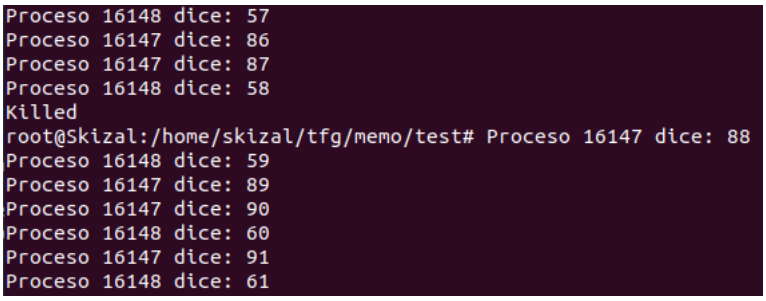


Figura 37: *Estado de contadores al restaurar.*

#### 9.2.2.3 Escritor

En un terminal hemos ejecutado:

- sudo singularity shell bionicCRIU.sif
- ./manager.sh $pipe &
- ./escritor.sh

El resto de comandos se ejecutan desde otro terminal con permisos de root.

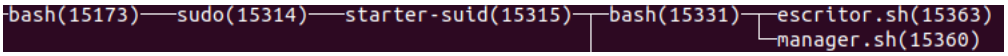


Figura 38: *Árbol de procesos escritor y manager.*

Ejecutamos: ./checkpoint.sh $pipe $directorioImagenes $PIDEscritor 0

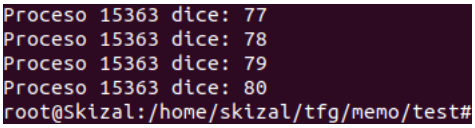


Figura 39: *Estado de escritor antes del checkpoint.*

Ejecutamos: ./restore.sh $pipe $directorioImagenes 0

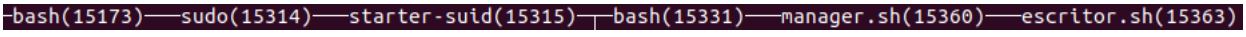


Figura 40: *Árbol de procesos escritor y manager al restaurar.*

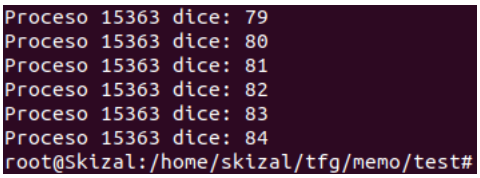


Figura 41: *Estado de escritor al restaurar.*

#### 9.2.2.4 Lector

En un terminal hemos ejecutado:

- sudo singularity shell bionicCRIU.sif
- ./manager.sh $pipe &
- ./lector.sh $fichero

El resto de comandos se ejecutan desde otro terminal con permisos de root.

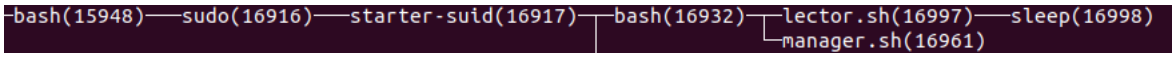


Figura 42: *Árbol de procesos lector y manager.*

Ejecutamos: ./checkpoint.sh $pipe $directorioImagenes $PIDLector 0

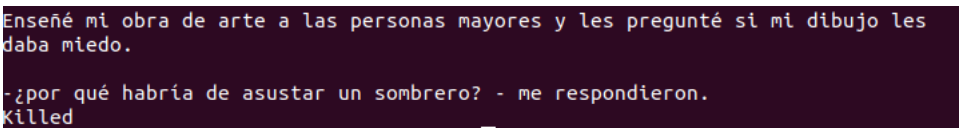


Figura 43: *Estado de lector antes del checkpoint.*

Ejecutamos: ./restore.sh $pipe $directorioImagenes 0

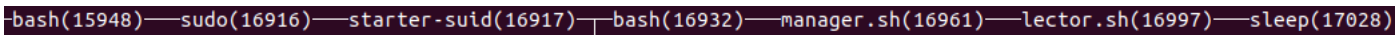


Figura 44: *Árbol de procesos lector y manager al restaurar.*

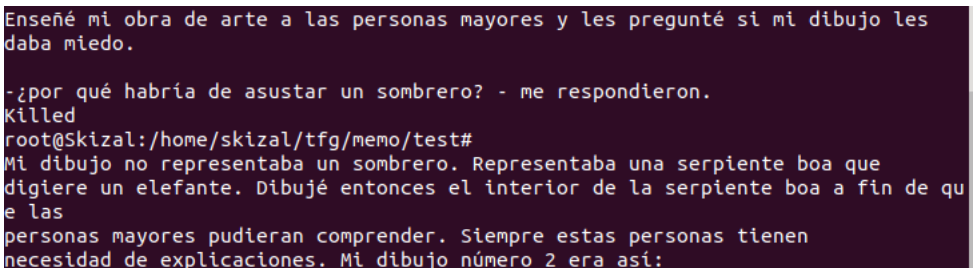


Figura 45: *Estado de lector al restaurar.*

#### 9.2.2.5 Cliente/Servidor TCP Contador

En un terminal hemos ejecutado:

- sudo singularity shell bionicCRIU.sif
- ./manager.sh $pipe &
- ./criuTCP 5000

Desde otro terminal ejecutamos: ./criuTCP 127.0.0.1 5000 El resto de comandos se ejecutan desde otro terminal con permisos de root.


Figura 46: *Árbol de procesos criuTCP (servidor) y manager.*

Ejecutamos: ./checkpoint.sh $pipe $directorioImagenes $PIDcriuTCPPadre 2

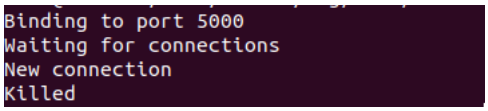


Figura 47: *Estado de criuTCP(servidor) antes del checkpoint.*

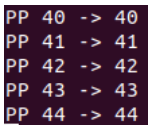


Figura 48: *Estado de criuTCP(cliente) antes del checkpoint.*

Ejecutamos: ./restore.sh $pipe $directorioImagenes 2

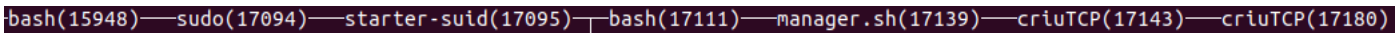


Figura 49: *Árbol de procesos criuTCP(servidor) y manager al restaurar.*

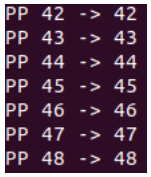


Figura 50: *Estado de criuTCP(cliente) al restaurar.*

#### 9.2.2.6 Manager y contenedores en instancias

En un terminal hemos ejecutado:

- sudo singularity instance start bionicCRIU.sif bionic
- sudo singularity run –app manager instace://bionic &
- sudo singularity run –app sinContador instance://bionic &

El resto de comandos se ejecutan desde otro terminal con permisos de root.

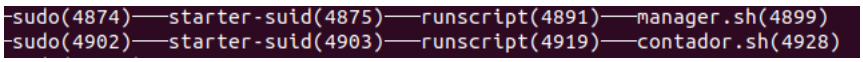


Figura 51: *Árbol de procesos contador y manager, instancia.*

Ejecutamos: ./checkpoint.sh $pipe $directorioImagenes $PIDcontador 0

En este caso el $PIDcontador ha de ser el PID que se ve desde dentro de la instancia. Nuestro contador ya nos dice su PID, pero si no lo hiciera, para verlo podemos ejecutar:

- sudo singularity shell instance://bionic
- ps aux

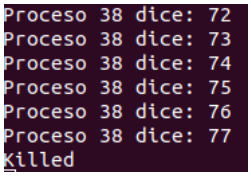


Figura 52: *Estado de contador antes del checkpoint, instancia.*

Ejecutamos: ./restore.sh $pipe $directorioImagenes 0

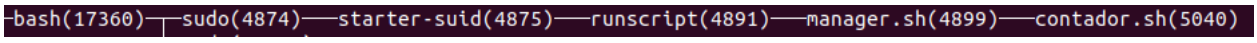


Figura 53: *Árbol de procesos contador y manager al restaurar, instancia.*

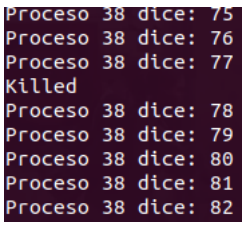


Figura 54: *Estado de contador al restaurar, instancia.*

# 10 Conclusiones

En este proyecto presentamos tecnologías bastante novedosas como son los contenedores software Singularity y CRIU. Una parte muy atractiva del proyecto era que estábamos trabajando en algo totalmente nuevo con tecnologías de actualidad, la integración de CRIU en Singularity.

El hecho de que fuera algo tan nuevo ha dificultado la resolución de problemas que iban surgiendo a lo largo del proyecto. La literatura sobre CRIU en Singularity es prácticamente nula y aunque CRIU se ha integrado en otras tecnologías de contenedores, ciertos conceptos no se trasladaban a Singularity. Lo que ha hecho que el objetivo del proyecto acabara siendo modificado.

El objetivo inicial del proyecto se ha cumplido de manera parcial. Hemos desarrollado una solución mediante la cual hemos podido realizar el *checkpoint/restore* de aplicaciones desde dentro del contenedor.

Nuestra solución final tiene algunos inconvenientes como la necesidad de tener privilegios de root, la ejecución del manager dentro del contenedor y la necesidad de utilizar directorios visibles desde el anfitrión y el contenedor, para poder comunicarnos mediante la pipe.

Aun así, si comparamos nuestra solución con la solución ideal de nuestro objetivo inicial, el *checkpoint/restore* del contenedor en su totalidad, vemos que tenemos un grado de usabilidad semejante.

No tenemos el *checkpoint* del contenedor entero pero tenemos el de la aplicación. Al final, cuando hacemos *checkpoint* del contenedor, lo hacemos del entorno de aislamiento (el contenedor), que no debería cambiar, más las aplicaciones que se ejecuten dentro. Ese entorno ya lo tenemos en la imagen del contenedor y para emular la restauración del contenedor sólo tenemos que:

- Crear el contenedor a partir de la misma imagen en la que se estaba ejecutando nuestra aplicación.
- Ejecutar dentro del contenedor nuestro manager y enviar el comando necesario para que se restaure la aplicación.

De esta manera el resultado final es parecido al previsto inicialmente, se ha restaurado la aplicación dentro del mismo contenedor Singularity.

De cara al futuro, sería interesante que Singularity añadiera algún modo de comunicarse con el proceso *starter-suid*. Para que, además de monitorizar, fuera capaz de ejecutar el *checkpoint/restore* y que hubiera un campo en el SIF en el que se pudieran guardar las imágenes de estado de las aplicaciones. Eso eliminaría la necesidad de usar un programa externo como el manager y mantendría la portabilidad de Singularity intacta, al seguir teniendo toda la información contenida en un único fichero imagen SIF.

Tengo una valoración final positiva del proyecto. Aún habiendo sido un proyecto que se ha encontrado con obstáculos, retrasos y cambios, se ha conseguido sacar adelante una solución para nuestro problema y he aprendido valiosas lecciones para el futuro.

# 11 Glosario

- **Máquina virtual**: Software que simula un sistema de computación y puede ejecutar programas como si fuese una computadora real.
- **ADC (Application Delivery Controller)**: Es un dispositivo que se coloca en un centro de datos entre el firewall y uno o más servidores de aplicaciones. Normalmente realizan la aceleración de aplicaciones y gestiona el balanceo de carga entre servidores.
- **HPC (High Performance Computing)**: La computación de alto rendimiento es la agregación de potencia de cálculo para resolver problemas complejos.
- **GPU (Graphical Processing Unit)**: Es un coprocesador dedicado al procesamiento de gráficos u operaciones de coma flotante.
- **Infiniband**: Es un bus de comunicaciones serie de alta velocidad, baja latencia y de baja sobrecarga de CPU, diseñado tanto para conexiones internas como externas.
- **Lustre**: Es un sistema de ficheros paralelo de código libre, que soporta muchos requisitos de entornos de simulación HPC.
- **MPI (Message Passing Interface)**: Es un estándar que define la sintaxis y la semántica de las funciones contenidas en una biblioteca de paso de mensajes diseñada para ser usada en programas que exploten la existencia de múltiples procesadores.
- **Control Groups**: Es una funcionalidad del núcleo de Linux que limita, mantiene la cuenta y aísla los recursos usados de un conjunto de procesos.
- **BLOB, Binary Large Object**: Es un elemento grande de datos que está en código binario.
- **VMA, Virtual Memory Area**: Es un rango contiguo de direcciones de memoria virtual utilizadas para acceder a la memoria física del sistema.
- **PIE, Position-Independent Executable**: Es un ejecutable que se ejecuta de manera correcta independientemente de su dirección absoluta de memoria.
- **Daemon**: Es un tipo especial de proceso informático que se ejecuta en segundo plano. No suele disponer de una "interfaz" directa con el usuario y no hacen uso de las entradas y salidas estándar para la comunicación.

# 12 Anexo A: Singularity Definition Files

```
BootStrap: debootstrap
OSVersion: trusty
MirrorURL: http://us.archive.ubuntu.com/ubuntu/

%environment
    SINGULARITY_SHELL=/bin/bash

%runscript
    /home/skizal/tfg/memo/test/contador.sh
```

Figura 55: *Definition file del contenedor trusty.sif.*

```
BootStrap: debootstrap
OSVersion: bionic
MirrorURL: http://us.archive.ubuntu.com/ubuntu/

%post
    apt-get update -y
    apt-get dist-upgrade -y
    apt-get install software-properties-common -y
    add-apt-repository 'deb http://es.archive.ubuntu.com/ubuntu/ bionic universe'
    apt-get update -y
    apt-get dist-upgrade -y
    apt-get install build-essential -y
    apt-get install criu -y
    apt-get install iptables -y

%environment
    SINGULARITY_SHELL=/bin/bash

%apprun manager
    /home/skizal/tfg/memo/test/manager.sh dumps/pipe

%apprun sidContador
    setsid /home/skizal/tfg/memo/test/contador.sh
```

Figura 56: *Definition file del contenedor bionicCRIU.sif.*

# 13 Anexo B: Scripts

```
#!/bin/bash

usage(){
    echo "Usage:"
    echo -e "\nUsage: ./manager.sh [pipe] &"
    echo -e "\t[pipe]: Location of pipe listening in the container."
}

if [[ $# -lt 1 ]]; then
    echo "Missing argument [pipe]."
    usage
    exit 1
fi

pipe=$1

if [[ -p $pipe ]]; then
    rm -f $pipe
    mkfifo $pipe
fi

if [[ ! -p $pipe ]]; then
    mkfifo $pipe
fi


while true
do
    if read line < "${pipe}"; then
        if [[ "$line" == 'quit' ]]; then
            exit 1
        fi
        eval $line 2> warningManager.txt
    fi
done

echo "Reader exiting"
```

Figura 57: *Manager.sh*

```
#!/bin/bash
usage(){
    echo "This script must be run by root: sudo -i"
    echo -e "\nUsage: ./checkpoint.sh [pipe] [checkpointLocation] [pid] [type]"
    echo -e "\t[pipe]: Location of pipe listening in the container."
    echo -e "\t[checkpointLocation]: Location with where criu will create image files.
    Must be bind mounted from host."
    echo -e "\t[pid]:  Process identifier number to checkpoint."
    echo -e "\t[type]: Type of process where:"
    echo -e "\t\t 0: process executed from a shell."
    echo -e "\t\t 1: process with TCP connection."
    echo -e "\t\t 2: 0 and 1."
    echo -e "\t\t 3: 0 and --leave-running"
    echo -e "\t\t 4: checkpoint process tree and kill manager."
}
if [[ $# -le 3 ]]; then
    args=4
    echo "Missing arguments $(($args-$#))."
    usage
    exit 1
fi
if [[ $4 -gt 4 || $4 -lt 0 ]]; then
    echo "Incorrect type value $3."
    exit 1
fi
if [[ ! -p $1 ]]; then
    echo "Pipe: $1 doesn't exist."
    exit 1
fi

shell="--shell-job"
tcp="--tcp-established"
if [[ $4 -eq 0 ]]; then
    echo "criu dump -o dump.log -v --images-dir $2 -t $3 $shell" >$1
fi
if [[ $4 -eq 1 ]]; then
    echo "criu dump -o dump.log -v --images-dir $2 -t $3 $tcp" >$1
fi
if [[ $4 -eq 2 ]]; then
    echo "criu dump -o dump.log -v --images-dir $2 -t $3 $shell $tcp" >$1
fi
if [[ $4 -eq 3 ]]; then
    echo "criu dump -o dump.log -v --images-dir $2 -t $3 $shell --leave-running" >$1
fi
if [[ $4 -eq 4 ]]; then
    echo "criu dump -o dump.log -v --images-dir $2 -t $3 $shell " >$1
    sleep 1
    echo "quit" > $1
fi
```



Figura 58: *Checkpoint.sh*

```
#!/bin/bash

usage(){
echo "This script must be run by root: sudo -i"
echo -e "\nUsage: ./restore.sh [pipe] [restoreLocation] [type]"
echo -e "\t[pipe]: Location of pipe listening in the container."
echo -e "\t[restoreLocation]: Location with criu files need for restore."
echo -e "\t[type]: Type of process where:"
echo -e "\t\t 0: process executed from a shell."
echo -e "\t\t 1: process with TCP connection."
echo -e "\t\t 2: 0 and 1."
}


if [[ $# -le 1 ]]; then
args=3
echo "Missing arguments $(($args-$#))."
usage
exit 1
fi
if [[ $3 -gt 2 || $3 -lt 0 ]]; then
echo "Incorrect type value $3."
exit 1
fi
if [[ ! -p $1 ]]; then
    echo "Pipe: $1 doesn't exist."
    exit 1
fi

shell="--shell-job"
tcp="--tcp-established"

if [[ $3 -eq 0 ]]; then
    echo "criu restore -o restore.log -v --images-dir $2 $shell --restore-detached
    --restore-sibling" >$1
fi
if [[ $3 -eq 1 ]]; then
    echo "criu restore -o restore.log -v --images-dir $2 $tcp --restore-detached
    --restore-sibling" >$1
fi
if [[ $3 -eq 2 ]]; then
    echo "criu restore -o restore.log -v --images-dir $2 $shell $tcp --restore-detached
    --restore-sibling" >$1
fi
```

Figura 59: *Restore.sh.*

```
#!/bin/bash

function_to_fork() {
    let C=1
    let D=1
    while true ; do
     COUNT=1000000
     while (( COUNT > 0 )); do
     (( COUNT -- ))
     done

     C=$((C + D))

     echo "Proceso $$ dice: $C"
    done
}

function_to_fork2() {
    let C=1
    let D=1
    while true ; do
     COUNT=1500000
     while (( COUNT > 0 )); do
     (( COUNT -- ))
     done

     C=$((C + D))

     echo "Proceso $! dice: $C"
    done
}

echo "Creacion de procesos"
function_to_fork &
function_to_fork2
```

Figura 60: *Contadores.sh.*

```
#!/bin/bash

let C=1
let D=1
while true ; do
    COUNT=1000000
    while (( COUNT > 0 )); do
     (( COUNT -- ))
    done

    C=$((C + D))

    echo "Proceso $$ dice: $C"
done
```

Figura 61: *Contador.sh.*

```
#!/bin/bash

let C=1
let D=1
while true ; do
COUNT=1000000
while (( COUNT > 0 )); do
(( COUNT -- ))
done

C=$((C + D))

echo "Proceso $$ dice: $C " >> outputEscritor.txt
done
```

Figura 62: *Escritor.sh.*

```
#!/bin/bash
FILE=$1
while read LINE; do
sleep 3
  echo "$LINE"
done < $FILE
```

Figura 63: *Lector.sh.*

# 14 Anexo C: Debug de Singularity

```
VERBOSE [U=0,P=21817]      print()                     Set messagelevel to: 5
DEBUG   [U=0,P=21817]      init()                      PIPE_EXEC_FD value: 7
VERBOSE [U=0,P=21817]      init()                      Container runtime
VERBOSE [U=0,P=21817]      is_suid()                   Check if we are running as setuid
DEBUG   [U=0,P=21817]      init()                      Overlay seems supported by kernel
DEBUG   [U=0,P=21817]      init()                      Drop privileges
DEBUG   [U=1000,P=21817]   init()                      Read json configuration from pipe
DEBUG   [U=1000,P=21817]   init()                      Set child signal mask
DEBUG   [U=1000,P=21817]   init()                      Wait completion of stage1
VERBOSE [U=1000,P=21826]   priv_escalate()             Get root privileges
DEBUG   [U=0,P=21826]      set_parent_death_signal()   Set parent death signal to 9
DEBUG   [U=0,P=21826]      prepare_stage()             Entering in stage 1
DEBUG   [U=1000,P=21826]   set_parent_death_signal()   Set parent death signal to 9
VERBOSE [U=1000,P=21826]   init()                      Spawn stage 1
VERBOSE [U=1000,P=21826]   startup()                   Execute stage 1
DEBUG   [U=1000,P=21826]   Stage()                     Entering stage 1
DEBUG   [U=1000,P=21826]   Init()                      Entering image format intializer
DEBUG   [U=1000,P=21826]   Init()                      Check for image format sif
DEBUG   [U=1000,P=21817]   init()                      Create socketpair for master communication channel
DEBUG   [U=1000,P=21817]   cleanup_fd()                Check file descriptor /proc/self/fd/3 pointing to /home/skizal/tfg/memo/test/trusty.sif
DEBUG   [U=1000,P=21817]   cleanup_fd()                Check file descriptor /proc/self/fd/4 pointing to anon_inode:[eventpoll]
DEBUG   [U=1000,P=21817]   cleanup_fd()                Closing /proc/self/fd/4
VERBOSE [U=1000,P=21817]   priv_escalate()             Get root privileges
VERBOSE [U=0,P=21817]      create_namespace()          Create mount namespace
DEBUG   [U=0,P=21817]      init()                      Create RPC socketpair for communication between stage 2 and RPC server
DEBUG   [U=0,P=21817]      set_terminal_control()      Pass terminal control to child
VERBOSE [U=0,P=21817]      init()                      Spawn master process
DEBUG   [U=0,P=21834]      set_parent_death_signal()   Set parent death signal to 9
VERBOSE [U=0,P=21834]      create_namespace()          Create mount namespace
DEBUG   [U=0,P=21834]      set_parent_death_signal()   Set parent death signal to 9
DEBUG   [U=0,P=21834]      prepare_stage()             Entering in stage 2
VERBOSE [U=0,P=21835]      init()                      Spawn RPC server
DEBUG   [U=1000,P=21834]   set_parent_death_signal()   Set parent death signal to 9
VERBOSE [U=1000,P=21817]   startup()                   Execute master process
VERBOSE [U=0,P=21835]      startup()                   Serve RPC requests
DEBUG   [U=1000,P=21817]   setupSessionLayout()        Attempting to use overlayfs (enable overlay = try)
DEBUG   [U=1000,P=21817]   setupOverlayLayout()        Creating overlay SESSIONDIR layout
DEBUG   [U=1000,P=21817]   addRootfsMount()            Mount rootfs in read-only mode
DEBUG   [U=1000,P=21817]   addRootfsMount()            Mounting block [squashfs] image: /home/skizal/tfg/memo/test/trusty.sif
DEBUG   [U=1000,P=21817]   addKernelMount()            Checking configuration file for 'mount proc'
DEBUG   [U=1000,P=21817]   addKernelMount()            Adding proc to mount list
VERBOSE [U=1000,P=21817]   addKernelMount()            Default mount: /proc:/proc
DEBUG   [U=1000,P=21817]   addKernelMount()            Checking configuration file for 'mount sys'
DEBUG   [U=1000,P=21817]   addKernelMount()            Adding sysfs to mount list
VERBOSE [U=1000,P=21817]   addKernelMount()            Default mount: /sys:/sys
DEBUG   [U=1000,P=21817]   addDevMount()               Checking configuration file for 'mount dev'
DEBUG   [U=1000,P=21817]   addDevMount()               Adding dev to mount list
VERBOSE [U=1000,P=21817]   addDevMount()               Default mount: /dev:/dev
DEBUG   [U=1000,P=21817]   addHostMount()              Not mounting host file systems per configuration
VERBOSE [U=1000,P=21817]   addBindsMount()             Found 'bind path' = /etc/localtime, /etc/localtime
VERBOSE [U=1000,P=21817]   addBindsMount()             Found 'bind path' = /etc/hosts, /etc/hosts
DEBUG   [U=1000,P=21817]   addHomeStagingDir()         Staging home directory (/home/skizal) at /usr/local/var/singularity/mnt/session/home/skizal
DEBUG   [U=1000,P=21817]   addHomeMount()              Adding home directory mount [/usr/local/var/singularity/mnt/session/home/skizal:/home/skizal] to list using layer: overlay
DEBUG   [U=1000,P=21817]   isLayerEnabled()            Using Layer system: overlay
DEBUG   [U=1000,P=21817]   addTmpMount()               Checking for 'mount tmp' in configuration file
VERBOSE [U=1000,P=21817]   addTmpMount()               Default mount: /tmp:/tmp
VERBOSE [U=1000,P=21817]   addTmpMount()               Default mount: /var/tmp:/var/tmp
DEBUG   [U=1000,P=21817]   addScratchMount()           Not mounting scratch directory: Not requested
VERBOSE [U=1000,P=21817]   addCwdMount()               Default mount: /home/skizal/tfg/memo/test: to the container
DEBUG   [U=1000,P=21817]   addLibsMount()              Checking for 'user bind control' in configuration file
DEBUG   [U=1000,P=21817]   addResolvConfMount()        Adding /etc/resolv.conf to mount list
VERBOSE [U=1000,P=21817]   addResolvConfMount()        Default mount: /etc/resolv.conf:/etc/resolv.conf
```

Figura 64: *Primera parte de la información debug generada por singularity -d shell $imagen.*

```
DEBUG   [U=1000,P=21817]   addHostnameMount()           Skipping hostname mount, not virtualizing UTS namespace on user request
DEBUG   [U=1000,P=21817]   create()                     Mount all
DEBUG   [U=1000,P=21817]   mountGeneric()               Mounting tmpfs to /usr/local/var/singularity/mnt/session
DEBUG   [U=1000,P=21817]   mountImage()                 Mounting loop device /dev/loop1 to /usr/local/var/singularity/mnt/session/rootfs
DEBUG   [U=1000,P=21817]   mountGeneric()               Mounting overlay to /usr/local/var/singularity/mnt/session/final
DEBUG   [U=1000,P=21817]   setPropagationMount()        Set RPC mount propagation flag to SLAVE
VERBOSE [U=1000,P=21817]   Passwd()                     Checking for template passwd file: /usr/local/var/singularity/mnt/session/rootfs/etc/passwd
VERBOSE [U=1000,P=21817]   Passwd()                     Creating passwd content
VERBOSE [U=1000,P=21817]   Passwd()                     Creating template passwd file and appending user data: /usr/local/var/singularity/mnt/session/rootfs/etc/passwd
DEBUG   [U=1000,P=21817]   addIdentityMount()           Adding /etc/passwd to mount list
VERBOSE [U=1000,P=21817]   addIdentityMount()           Default mount: /etc/passwd:/etc/passwd
VERBOSE [U=1000,P=21817]   Group()                      Checking for template group file: /usr/local/var/singularity/mnt/session/rootfs/etc/group
VERBOSE [U=1000,P=21817]   Group()                      Creating group content
DEBUG   [U=1000,P=21817]   addIdentityMount()           Adding /etc/group to mount list
VERBOSE [U=1000,P=21817]   addIdentityMount()           Default mount: /etc/group:/etc/group
DEBUG   [U=1000,P=21817]   mountGeneric()               Remounting /usr/local/var/singularity/mnt/session/final
DEBUG   [U=1000,P=21817]   mountGeneric()               Mounting /dev to /usr/local/var/singularity/mnt/session/final/dev
DEBUG   [U=1000,P=21817]   mountGeneric()               Mounting /etc/localtime to /usr/local/var/singularity/mnt/session/final/etc/localtime
DEBUG   [U=1000,P=21817]   mountGeneric()               Mounting /etc/hosts to /usr/local/var/singularity/mnt/session/final/etc/hosts
DEBUG   [U=1000,P=21817]   mountGeneric()               Mounting /usr/local/etc/singularity/actions to /usr/local/var/singularity/mnt/session/final/.singularity.d/actions
DEBUG   [U=1000,P=21817]   mountGeneric()               Remounting /usr/local/var/singularity/mnt/session/final/.singularity.d/actions
DEBUG   [U=1000,P=21817]   mountGeneric()               Mounting /proc to /usr/local/var/singularity/mnt/session/final/proc
DEBUG   [U=1000,P=21817]   mountGeneric()               Remounting /usr/local/var/singularity/mnt/session/final/proc
DEBUG   [U=1000,P=21817]   mountGeneric()               Mounting sysfs to /usr/local/var/singularity/mnt/session/final/sys
DEBUG   [U=1000,P=21817]   mountGeneric()               Mounting /home/skizal to /usr/local/var/singularity/mnt/session/home/skizal
DEBUG   [U=1000,P=21817]   mountGeneric()               Remounting /usr/local/var/singularity/mnt/session/home/skizal
DEBUG   [U=1000,P=21817]   mountGeneric()               Mounting /usr/local/var/singularity/mnt/session/home/skizal to /usr/local/var/singularity/mnt/session/final/home/skizal
DEBUG   [U=1000,P=21817]   mountGeneric()               Remounting /usr/local/var/singularity/mnt/session/final/home/skizal
DEBUG   [U=1000,P=21817]   mountGeneric()               Mounting /tmp to /usr/local/var/singularity/mnt/session/final/tmp
DEBUG   [U=1000,P=21817]   mountGeneric()               Remounting /usr/local/var/singularity/mnt/session/final/tmp
DEBUG   [U=1000,P=21817]   mountGeneric()               Mounting /var/tmp to /usr/local/var/singularity/mnt/session/final/var/tmp
DEBUG   [U=1000,P=21817]   mountGeneric()               Remounting /usr/local/var/singularity/mnt/session/final/var/tmp
DEBUG   [U=1000,P=21817]   mountGeneric()               Skipping mount /usr/local/var/singularity/mnt/session/final/home/skizal/tfg/memo/test, /usr/local/var/singularity/mnt/session/final/home/
skizal already mounted
DEBUG   [U=1000,P=21817]   mountGeneric()               Mounting /usr/local/var/singularity/mnt/session/etc/resolv.conf to /usr/local/var/singularity/mnt/session/final/etc/resolv.conf
DEBUG   [U=1000,P=21817]   mountGeneric()               Mounting /usr/local/var/singularity/mnt/session/etc/passwd to /usr/local/var/singularity/mnt/session/final/etc/passwd
DEBUG   [U=1000,P=21817]   mountGeneric()               Mounting /usr/local/var/singularity/mnt/session/etc/group to /usr/local/var/singularity/mnt/session/final/etc/group
DEBUG   [U=1000,P=21817]   create()      Chroot into /usr/local/var/singularity/mnt/session/final
DEBUG   [U=0,P=21835]      Chroot()      Change current directory to /usr/local/var/singularity/mnt/session/final
DEBUG   [U=0,P=21835]      Chroot()                     Hold reference to host / directory
DEBUG   [U=0,P=21835]      Chroot()                     Called pivot_root on /usr/local/var/singularity/mnt/session/final
DEBUG   [U=0,P=21835]      Chroot()                     Change current directory to host / directory
DEBUG   [U=0,P=21835]      Chroot()                     Apply slave mount propagation for host / directory
DEBUG   [U=0,P=21835]      Chroot()                     Called unmount(/, syscall.MNT_DETACH)
DEBUG   [U=0,P=21835]      Chroot()                     Changing directory to / to avoid getpwd issues
DEBUG   [U=1000,P=21817]   create()                     Chdir into / to avoid errors
VERBOSE [U=1000,P=21834]   startup()                    Execute stage 2
DEBUG   [U=1000,P=21834]   Stage()                      Entering stage 2
DEBUG   [U=1000,P=21817]   PostStartProcess()           Post start process
```

Figura 65: *Segunda parte de la información debug generada por singularity -d shell $imagen.*

```
VERBOSE [U=0,P=21904]      print()                      Set messagelevel to: 5
DEBUG   [U=0,P=21904]      init()                       PIPE_EXEC_FD value: 9
VERBOSE [U=0,P=21904]      init()                       Container runtime
VERBOSE [U=0,P=21904]      is_suid()                    Check if we are running as setuid
DEBUG   [U=0,P=21904]      init()                       Overlay seems supported by kernel
DEBUG   [U=0,P=21904]      init()                       Drop privileges
DEBUG   [U=1000,P=21904]   init()                       Read json configuration from pipe
DEBUG   [U=1000,P=21904]   init()                       Set child signal mask
DEBUG   [U=1000,P=21904]   init()                       Wait completion of stage1
VERBOSE [U=1000,P=21905]   priv_escalate()              Get root privileges
DEBUG   [U=0,P=21905]      set_parent_death_signal()    Set parent death signal to 9
DEBUG   [U=0,P=21905]      prepare_stage()              Entering in stage 1
DEBUG   [U=1000,P=21905]   set_parent_death_signal()    Set parent death signal to 9
VERBOSE [U=1000,P=21905]   init()                       Spawn stage 1
VERBOSE [U=1000,P=21905]   startup()                    Execute stage 1
DEBUG   [U=1000,P=21905]   Stage()                      Entering stage 1
DEBUG   [U=1000,P=21905]   Init()                       Entering image format intializer
DEBUG   [U=1000,P=21905]   Init()                       Check for image format sif
DEBUG   [U=1000,P=21904]   init()                       Create socketpair for master communication channel
VERBOSE [U=1000,P=21904]   init()                       Run as instance
DEBUG   [U=1000,P=21912]   cleanup_fd()                 Check file descriptor /proc/self/fd/3 pointing to /home/skizal/tfg/memo/test/trusty.sif
DEBUG   [U=1000,P=21912]   cleanup_fd()                 Check file descriptor /proc/self/fd/4 pointing to anon_inode:[eventpoll]
DEBUG   [U=1000,P=21912]   cleanup_fd()                 Closing /proc/self/fd/4
VERBOSE [U=1000,P=21912]   priv_escalate()              Get root privileges
VERBOSE [U=0,P=21912]      create_namespace()           Create mount namespace
DEBUG   [U=0,P=21912]      init()                       Create RPC socketpair for communication between stage 2 and RPC server
VERBOSE [U=0,P=21912]      pid_namespace_init()         Create pid namespace
VERBOSE [U=0,P=21912]      init()                       Spawn master process
DEBUG   [U=0,P=1]          set_parent_death_signal()    Set parent death signal to 9
VERBOSE [U=0,P=1]          create_namespace()           Create ipc namespace
VERBOSE [U=0,P=1]          create_namespace()           Create mount namespace
DEBUG   [U=0,P=1]          set_parent_death_signal()    Set parent death signal to 9
DEBUG   [U=0,P=1]          prepare_stage()              Entering in stage 2
DEBUG   [U=1000,P=1]       set_parent_death_signal()    Set parent death signal to 9
VERBOSE [U=0,P=2]          init()                       Spawn RPC server
VERBOSE [U=0,P=2]          startup()                    Serve RPC requests
VERBOSE [U=1000,P=21912]   startup()                    Execute master process
DEBUG   [U=1000,P=21912]   setupSessionLayout()         Attempting to use overlayfs (enable overlay = try)
DEBUG   [U=1000,P=21912]   setupOverlayLayout()         Creating overlay SESSIONDIR layout
DEBUG   [U=1000,P=21912]   addRootfsMount()             Mount rootfs in read-only mode
DEBUG   [U=1000,P=21912]   addRootfsMount()             Mounting block [squashfs] image: /home/skizal/tfg/memo/test/trusty.sif
DEBUG   [U=1000,P=21912]   addKernelMount()             Checking configuration file for 'mount proc'
DEBUG   [U=1000,P=21912]   addKernelMount()             Adding proc to mount list
VERBOSE [U=1000,P=21912]   addKernelMount()             Default mount: /proc:/proc
DEBUG   [U=1000,P=21912]   addKernelMount()             Checking configuration file for 'mount sys'
DEBUG   [U=1000,P=21912]   addKernelMount()             Adding sysfs to mount list
VERBOSE [U=1000,P=21912]   addKernelMount()             Default mount: /sys:/sys
DEBUG   [U=1000,P=21912]   addDevMount()                Checking configuration file for 'mount dev'
DEBUG   [U=1000,P=21912]   addDevMount()                Adding dev to mount list
VERBOSE [U=1000,P=21912]   addDevMount()                Default mount: /dev:/dev
DEBUG   [U=1000,P=21912]   addHostMount()               Not mounting host file systems per configuration
VERBOSE [U=1000,P=21912]   addBindsMount()              Found 'bind path' = /etc/localtime, /etc/localtime
VERBOSE [U=1000,P=21912]   addBindsMount()              Found 'bind path' = /etc/hosts, /etc/hosts
DEBUG   [U=1000,P=21912]   addHomeStagingDir()          Staging home directory (/home/skizal) at /usr/local/var/singularity/mnt/session/home/skizal
DEBUG   [U=1000,P=21912]   addHomeMount()               Adding home directory mount [/usr/local/var/singularity/mnt/session/home/skizal:/home/skizal] to list using layer: overlay
DEBUG   [U=1000,P=21912]   isLayerEnabled()             Using Layer system: overlay
DEBUG   [U=1000,P=21912]   addTmpMount()                Checking for 'mount tmp' in configuration file
VERBOSE [U=1000,P=21912]   addTmpMount()                Default mount: /tmp:/tmp
VERBOSE [U=1000,P=21912]   addTmpMount()                Default mount: /var/tmp:/var/tmp
DEBUG   [U=1000,P=21912]   addScratchMount()            Not mounting scratch directory: Not requested
VERBOSE [U=1000,P=21912]   addCwdMount()                Default mount: /home/skizal/tfg/memo/test: to the container
DEBUG   [U=1000,P=21912]   addLibsMount()               Checking for 'user bind control' in configuration file
```

Figura 66: *Primera parte de la información debug generada por singularity -d instance start $imagen $nombreImagen.*

```
DEBUG   [U=1000,P=21912]   addResolvConfMount()          Adding /etc/resolv.conf to mount list
VERBOSE [U=1000,P=21912]   addResolvConfMount()          Default mount: /etc/resolv.conf:/etc/resolv.conf
DEBUG   [U=1000,P=21912]   addHostnameMount()            Skipping hostname mount, not virtualizing UTS namespace on user request
DEBUG   [U=1000,P=21912]   create()                      Mount all
DEBUG   [U=1000,P=21912]   mountGeneric()                Mounting tmpfs to /usr/local/var/singularity/mnt/session
DEBUG   [U=1000,P=21912]   mountImage()                  Mounting loop device /dev/loop1 to /usr/local/var/singularity/mnt/session/rootfs
DEBUG   [U=1000,P=21912]   mountGeneric()                Mounting overlay to /usr/local/var/singularity/mnt/session/final
DEBUG   [U=1000,P=21912]   setPropagationMount()         Set RPC mount propagation flag to SLAVE
VERBOSE [U=1000,P=21912]   Passwd()                      Checking for template passwd file: /usr/local/var/singularity/mnt/session/rootfs/etc/passwd
VERBOSE [U=1000,P=21912]   Passwd()                      Creating passwd content
VERBOSE [U=1000,P=21912]   Passwd()                      Creating template passwd file and appending user data: /usr/local/var/singularity/mnt/session/rootfs/etc/passwd
DEBUG   [U=1000,P=21912]   addIdentityMount()            Adding /etc/passwd to mount list
VERBOSE [U=1000,P=21912]   addIdentityMount()            Default mount: /etc/passwd:/etc/passwd
VERBOSE [U=1000,P=21912]   Group()                       Checking for template group file: /usr/local/var/singularity/mnt/session/rootfs/etc/group
VERBOSE [U=1000,P=21912]   Group()                       Creating group content
DEBUG   [U=1000,P=21912]   addIdentityMount()            Adding /etc/group to mount list
VERBOSE [U=1000,P=21912]   addIdentityMount()            Default mount: /etc/group:/etc/group
DEBUG   [U=1000,P=21912]   mountGeneric()                Remounting /usr/local/var/singularity/mnt/session/final
DEBUG   [U=1000,P=21912]   mountGeneric()                Mounting /dev to /usr/local/var/singularity/mnt/session/final/dev
DEBUG   [U=1000,P=21912]   mountGeneric()                Mounting /etc/localtime to /usr/local/var/singularity/mnt/session/final/etc/localtime
DEBUG   [U=1000,P=21912]   mountGeneric()                Mounting /etc/hosts to /usr/local/var/singularity/mnt/session/final/etc/hosts
DEBUG   [U=1000,P=21912]   mountGeneric()                Mounting /usr/local/etc/singularity/actions to /usr/local/var/singularity/mnt/session/final/.singularity.d/actions
DEBUG   [U=1000,P=21912]   mountGeneric()                Remounting /usr/local/var/singularity/mnt/session/final/.singularity.d/actions
DEBUG   [U=1000,P=21912]   mountGeneric()                Mounting proc to /usr/local/var/singularity/mnt/session/final/proc
DEBUG   [U=1000,P=21912]   mountGeneric()                Mounting sysfs to /usr/local/var/singularity/mnt/session/final/sys
DEBUG   [U=1000,P=21912]   mountGeneric()                Mounting /home/skizal to /usr/local/var/singularity/mnt/session/home/skizal
DEBUG   [U=1000,P=21912]   mountGeneric()                Remounting /usr/local/var/singularity/mnt/session/home/skizal
DEBUG   [U=1000,P=21912]   mountGeneric()                Mounting /usr/local/var/singularity/mnt/session/home/skizal to /usr/local/var/singularity/mnt/session/final/home/skizal
DEBUG   [U=1000,P=21912]   mountGeneric()                Remounting /usr/local/var/singularity/mnt/session/final/home/skizal
DEBUG   [U=1000,P=21912]   mountGeneric()                Mounting /tmp to /usr/local/var/singularity/mnt/session/final/tmp
DEBUG   [U=1000,P=21912]   mountGeneric()                Remounting /usr/local/var/singularity/mnt/session/final/tmp
DEBUG   [U=1000,P=21912]   mountGeneric()                Mounting /var/tmp to /usr/local/var/singularity/mnt/session/final/var/tmp
DEBUG   [U=1000,P=21912]   mountGeneric()                Remounting /usr/local/var/singularity/mnt/session/final/var/tmp
DEBUG   [U=1000,P=21912]   mountGeneric()                Skipping mount /usr/local/var/singularity/mnt/session/final/home/skizal/tfg/memo/test, /usr/local/var/singularity/mnt/session/final/home/
skizal already mounted
DEBUG   [U=1000,P=21912]   mountGeneric()                Mounting /usr/local/var/singularity/mnt/session/etc/resolv.conf to /usr/local/var/singularity/mnt/session/final/etc/resolv.conf
DEBUG   [U=1000,P=21912]   mountGeneric()                Mounting /usr/local/var/singularity/mnt/session/etc/passwd to /usr/local/var/singularity/mnt/session/final/etc/passwd
DEBUG   [U=1000,P=21912]   mountGeneric()                Mounting /usr/local/var/singularity/mnt/session/etc/group to /usr/local/var/singularity/mnt/session/final/etc/group
DEBUG   [U=1000,P=21912]   create()                      Chroot into /usr/local/var/singularity/mnt/session/final
DEBUG   [U=0,P=2]          Chroot()                      Change current directory to /usr/local/var/singularity/mnt/session/final
DEBUG   [U=0,P=2]          Chroot()                      Hold reference to host / directory
DEBUG   [U=0,P=2]          Chroot()                      Called pivot_root on /usr/local/var/singularity/mnt/session/final
DEBUG   [U=0,P=2]          Chroot()                      Change current directory to host / directory
DEBUG   [U=0,P=2]          Chroot()                      Apply slave mount propagation for host / directory
DEBUG   [U=0,P=2]          Chroot()                      Called unmount(/, syscall.MNT_DETACH)
DEBUG   [U=0,P=2]          Chroot()                      Changing directory to / to avoid getpwd issues
DEBUG   [U=1000,P=21912]   create()                      Chdir into / to avoid errors
VERBOSE [U=1000,P=1]       startup()                     Execute stage 2
DEBUG   [U=1000,P=1]       Stage()                       Entering stage 2
DEBUG   [U=1000,P=21912]   PostStartProcess()            Post start process

VERBOSE [U=1000,P=21896]   execStarter()                 you will find instance output here: /home/skizal/.singularity/instances/Skizal/skizal/trusty.out
VERBOSE [U=1000,P=21896]   execStarter()                 you will find instance error here: /home/skizal/.singularity/instances/Skizal/skizal/trusty.err
INFO    [U=1000,P=21896]   execStarter()                 instance started successfully
```

Figura 67: *Segunda parte de la información debug generada por singularity -d instance start $imagen $nombreImagen.*

```
VERBOSE [U=0,P=22358]      print()                       Set messagelevel to: 5
DEBUG   [U=0,P=22358]      init()                        PIPE_EXEC_FD value: 7
VERBOSE [U=0,P=22358]      init()                        Container runtime
VERBOSE [U=0,P=22358]      is_suid()                     Check if we are running as setuid
DEBUG   [U=0,P=22358]      init()                        Overlay seems supported by kernel
DEBUG   [U=0,P=22358]      init()                        Drop privileges
DEBUG   [U=1000,P=22358]   init()                        Read json configuration from pipe
DEBUG   [U=1000,P=22358]   init()                        Set child signal mask
DEBUG   [U=1000,P=22358]   init()                        Wait completion of stage1
VERBOSE [U=1000,P=22366]   priv_escalate()               Get root privileges
DEBUG   [U=0,P=22366]      set_parent_death_signal()     Set parent death signal to 9
DEBUG   [U=0,P=22366]      prepare_stage()               Entering in stage 1
DEBUG   [U=1000,P=22366]   set_parent_death_signal()     Set parent death signal to 9
VERBOSE [U=1000,P=22366]   init()                        Spawn stage 1
VERBOSE [U=1000,P=22366]   startup()                     Execute stage 1
DEBUG   [U=1000,P=22366]   Stage()                       Entering stage 1
DEBUG   [U=1000,P=22358]   init()                        Create socketpair for master communication channel
DEBUG   [U=1000,P=22358]   cleanup_fd()                  Check file descriptor /proc/self/fd/4 pointing to anon_inode:[eventpoll]
DEBUG   [U=1000,P=22358]   cleanup_fd()                  Closing /proc/self/fd/4
VERBOSE [U=1000,P=22358]   priv_escalate()               Get root privileges
VERBOSE [U=0,P=22358]      enter_namespace()             Entering in pid namespace
DEBUG   [U=0,P=22358]      enter_namespace()             Opening namespace file descriptor /proc/22292/root/run/singularity/instances/skizal/trusty/ns/pid
DEBUG   [U=0,P=22358]      set_terminal_control()        Pass terminal control to child
VERBOSE [U=0,P=22358]      init()                        Spawn master process
DEBUG   [U=0,P=26]         set_parent_death_signal()     Set parent death signal to 9
VERBOSE [U=0,P=26]         enter_namespace()             Entering in ipc namespace
DEBUG   [U=0,P=26]         enter_namespace()             Opening namespace file descriptor /proc/22292/root/run/singularity/instances/skizal/trusty/ns/ipc
VERBOSE [U=0,P=26]         enter_namespace()             Entering in mount namespace
DEBUG   [U=0,P=26]         enter_namespace()             Opening namespace file descriptor /proc/22292/root/run/singularity/instances/skizal/trusty/ns/mnt
VERBOSE [U=0,P=26]         init()                        Spawn stage 2
VERBOSE [U=0,P=26]         init()                        Don't execute RPC server, joining instance
DEBUG   [U=0,P=26]         set_parent_death_signal()     Set parent death signal to 9
DEBUG   [U=0,P=26]         prepare_stage()               Entering in stage 2
DEBUG   [U=1000,P=26]      set_parent_death_signal()     Set parent death signal to 9
DEBUG   [U=1000,P=22358]   init()                        Wait stage 2 child process
VERBOSE [U=1000,P=26]      startup()                     Execute stage 2
DEBUG   [U=1000,P=26]      Stage()                       Entering stage 2
```

Figura 68: *Información debug generada por singularity -d shell instance://$imagen.*

# 15 Referencias


[1] What are linux containers https://opensource.com/resources/what-are-linux-containers.

[2] What are containers https://www.sdxcentral.com/cloud/containers/definitions/what-are-containers-like-docker-linux-containers/.

[3] Singularity talk hpc https://www.youtube.com/watch?v=WweW0EUlh_0.

[4] Vm vs container https://www.sdxcentral.com/containers/definitions/containers-vs-vms.

[5] Industry Report. Container infrastructure – what you need to know in 2018.

[6] Singularity about https://singularity.lbl.gov/about.

[7] Hilo petición criu-singularity https://github.com/sylabs/singularity/issues/468.

[8] Xiao Chen, Jian Hui Jiang, and Qu Jiang. A method of self-adaptive pre-copy container checkpoint. *Proceedings - 2015 IEEE 21st Pacific Rim International Symposium on Dependable Computing, PRDC 2015*, pages 290–300, 2016.

[9] C/r container redhat https://www.redhat.com/en/blog/checkpointrestore-container-migration.

[10] Criu https://criu.org/Main_page.

[11] Alexey Vasyukov and Katerina Beklemysheva. Using criu with hpc containers : Field experience. 7(7):24106–24108, 2018.

[12] Criu limitation https://criu.org/What_cannot_be_checkpointed.

[13] Criu changes at cr https://criu.org/What_can_change_after_C/R.

[14] Integration criu https://criu.org/Integration.

[15] Glassdoor https://www.glassdoor.com.

[16] Tfg informe de sostenibilidad https://www.fib.upc.edu/sites/fib/files/documents/estudis/tfg-informe-sostenibilitat-2018.pdf.

[17] License singularity https://github.com/sylabs/singularity/blob/master/LICENSE.md.

[18] Singularity sif https://www.sylabs.io/2018/03/sif-containing-your-containers/.

[19] Guia de usuario de singularity https://www.sylabs.io/guides/3.0/user-guide/.

[20] *Definition Files* de singularity https://www.sylabs.io/guides/3.0/user-guide/definition_files.html.

[21] Linux namespaces https://lwn.net/Articles/531114/.

[22] Cgroups de linux https://wiki.archlinux.org/index.php/cgroups.

[23] Mount namespaces https://lwn.net/Articles/689856/.

[24] Pid namespaces https://lwn.net/Articles/259217/.

[25] Construcción de contenedores singularity https://www.sylabs.io/guides/3.0/user-guide/build_a_container.html.

[26] Instancias singularity https://www.sylabs.io/guides/3.0/user-guide/running_services.html.

[27] Código singularity monitor.go https://github.com/sylabs/singularity/blob/master/internal/pkg/runtime/engines/singularity/monitor.go.

[28] Código singularity engines_linux.go https://github.com/sylabs/singularity/blob/master/internal/pkg/runtime/engines/engines_linux.go.

[29] Docker https://docs.docker.com/engine/docker-overview/.

[30] Criu c/r internal slides https://es.slideshare.net/andreywagin/checkpointrestore-mostly-in-userspace-16408070.

[31] Criu c/r internal https://criu.org/Checkpoint/Restore.

[32] Funcionamiento freezer https://www.kernel.org/doc/Documentation/cgroup-v1/freezer-subsystem.txt.

[33] Funcionamiento ptrace https://lwn.net/Articles/441990/.

[34] Criu parasite https://criu.org/Parasite_code.

[35] Criu tree after restore https://criu.org/Tree_after_restore.

[36] Ejemplo de uso de criu con conexiones tcp https://criu.org/Simple_TCP_pair.